ETH ZURICH

BACHELOR THESIS

Neuronally-inspired path integration and map formation with visual correction realized in mixed-signal neuromorphic devices

Author:
David Niederberger

Supervisors:
Yulia Sandamirskaya
Angelika Steger

A thesis submitted in fulfillment of the requirements for the degree of BSc ETH in Computer Science

in the

Neuromorphic Cognitive Systems Group
Institute of Neuroinformatics

June 2018

Acknowledgements

I would like to thank my supervisor Yulia Sandamirskaya for the support and guidance along the development of this bachelor thesis. I'm glad I could do my thesis at the INI, although having only an computer science background.
I also want to thank Raphaela Kreiser for always taking the time to help me thorough the project and answering my questions.
Then I would like to thank Julien Martel for taking the time to explain me the code he wrote, which I used in my project.
My thanks go also to Angelika Steger, who agreed to be my supervisor from the department of computer science at ETH.
Furthermore, I would like to thank Giacomo Indiveri for letting me do my bachelors thesis in the NCS group at the INI.
Finally, I want to thanks my family and friends, who supported and encouraged me at all times.

Contents

1. Introduction

Robots, which operate autonomously in their environment are faced with many problems, that are solved by even the smallest animals. Amongst other, this contains the task of Simultaneous Localization and Mapping (SLAM), a crucial skill for animals as well as autonomous robots to move in an unknown environment. For a mobile robot, this problem begins, when it is placed at an initial point and start foraging in its environment. The robot receives some noisy sensory input and starts building a map, simultaneously it has to localize itself in this map. One part of SLAM is Path Integration, a process where the agent computes the direction and distance from his current location relative to a reference point by integrating self-motion and visual cues (Kreiser et al., 2018). A mobile robot might obtain these self-motion cues from its wheel encoders or from some motor commands, but this leads, besides the noisy sensory input, to another problem, namely that its motion is uncertain. There are many reasons for this, such as an uneven or slippery ground, or fabrication mismatch and due to the motion uncertainty, errors in the path integration process accumulate. To handle these errors, often a correction mechanisms over visual inputs is implemented.
In the field of autonomous robots, there is a lot of research to solve the SLAM problem purely mathematical, meaning that all internal and external information is processed with algorithms. The earliest SLAM algorithm uses one state vector and a covariance matrix to estimate the location and account for uncertainties. As the robot moves trough the new environment, it updates the information from the vector and matrix trough an extended kalman filter. When the robot recognizes a landmark, meaning a visual cue which the robot has already seen and knows its position relatively well, the error of the robots position estimate is reduced and also the uncertainty of other landmark positions decreases. Another mathematical approach to SLAM is through particle filters. Here, the state of the map and the location of the robot are guessed upon sensory information. These guesses are represented by particles and by collecting many of them, the true state can

be approached stochastically trough particle filters. A third group of SLAM algorithms models SLAM as a graph problem, where the previous robots positions and recognized landmarks are represented in a graph and the current position is estimated by nonlinear sparse optimizations (Siciliano, 2007, chap. 46).
Although there has been great progress in solving the SLAM problem over the last decades, it is still a major research topic in the field of robotics and alternative solving approaches have emerged (Cadena et al., 2016). Many inventions and technological improvements have been inspired by nature, especially in the field of robotics. In the research of biological mapping and navigation, rodents are one of the most studied animals. It has been established, that these animals form a cognitive map of their environment, meaning that they not only learn things related to the current task, but can later fall back to learned information if needed (Milford, 2008, chap. 4). Rats have different neuronal cells which encode spatial information, such as place cells and head direction cells (O'Keefe and Dostrovsky, 1971; O'Keefe and Conway, 1978). This encoding is achieved by assigning the cells to orientation or spatial location and letting the corresponding cells fire. Updating the spatial information is part of path integration the rat brain performs, which can accumulate errors, due to missing visual or other sensory input. With help of strong visual cues, these errors can be corrected (Taube et al., 1990).
Many of the biological theories about SLAM have been computationally modeled and tested. One of these models is the RatSLAM architecture (Milford et al., 2004), which connects a neuronal network with a mobile robot. In this experiment, the place cells of rats were emulated by an competitive attractor network structure named 'pose cells'. This network structure, also called Winner-take-all structure has the property, that each unit is connected excitatory with its neighboring units and inhibits those further away. The result is a convergence of the activity to a few units, representing in this case the position of the robot used. These pose cells were fed with information from internal sensors on one hand and with vision cues from external sensors on the other. With these informations, path integration was performed and could be corrected. The RatSLAM experiment was one of many models of neural networks, which proved that the SLAM problem can be tackled with help of neuronal computing.
These approaches to solve SLAM, whether they are based on mathematics or neural networks, have all one big drawback, they run on conventional computer platforms and consume therefore rather a lot of energy, which can be a problem in certain scenarios where mobile robots are used. Here, neuromorphic hardware

which can directly emulate the biophysics of neuronal networks, offers a low-power solution. Neuromorphic computers have their origins in the 1980s, when the electrical engineer Carver Mead researched about biologically inspired micro-electronics using analog VLSI (very large scale integration) circuits (Mead, 1989). Mead discovered, that CMOS transistors, which were commonly used in digital circuits as a binary on-and-off switch, have similar current-voltage relationship as neuronal ion channels when operating in a subthreshold regime, meaning that the current between the source and drain of the transistor is below the current threshold it needs to have an "on"-state (Poon and Zhou, 2011). Exploiting this subthreshold regime allows to emulate the electrophysical dynamics of real neurons directly on analog VLSI chips. In the last two decades, researches developed new hardware using these neuronally inspired analog VLSI circuits also called silicon neurons, enabling low-power, real-time emulating of spiking neural networks (Indiveri et al., 2011).

In this thesis, neuromorphic hardware, namely the ROLLS neuromorphic processor is used, to implement and test a spiking neural network, which models a head direction network inspired by biological orientation mechanisms observed in rodents. In addition to prior models of this network (Kreiser et al., 2018), a learning mechanism is implemented to enable learning the position of blinking lights as visual cues, which should minimize accumulating errors by resetting the orientation of the head direction with help of these visual cues.

2. Methods

In order to implement and test the neuronal network, different hardware and software pieces were used. The hardware contains the ROLLS neuromorphic device, the Parallella board and the "pushbot", a robot with embedded Dynamic Vision Sensor (eDVS) mounted on. In the hardware part (section 2.1), first, the different pieces are introduced, then the whole setup is described. Then a software part follows (section 2.2) and in the end the neuronal network architecture (section 2.3) is explained.

2.1 Hardware architecture

2.1.1 ROLLS

The reconfigurable on-line learning spiking neuromorphic processor (Qiao et al., 2015) is a neural processing system, which allows to emulate biophysics of real spiking neurons and synapses. It comprises a silicon neuron block, a plastic synapse array and a nonplastic synapse array.

The neuron block consists of 256 neuron circuits, which is derived from the adaptive exponential I&F (Integrate & Fire) circuit proposed in Indiveri et al. (2011). It allows to emulate a wide range of neural behaviors, such as spike-frequency adaptation properties, refractory period mechanism and adjustable spiking threshold mechanism. A on-chip programmable bias generator allows to set up to 13 different parameters to modify the behavior of the neuron circuits, however these parameters are the same for every neuron.
256 x 256 synapses form the plastic synapse array which can be used to set a learning behavior between neurons. The synapses either show a Long-Term Potentiation (LTP) oder a Long-Term Depression (LDP), meaning that they either send spikes from one neuron to another or do not. Its circuits model a learning algorithm, that implements the spike-driven synaptic plasticity rule proposed by

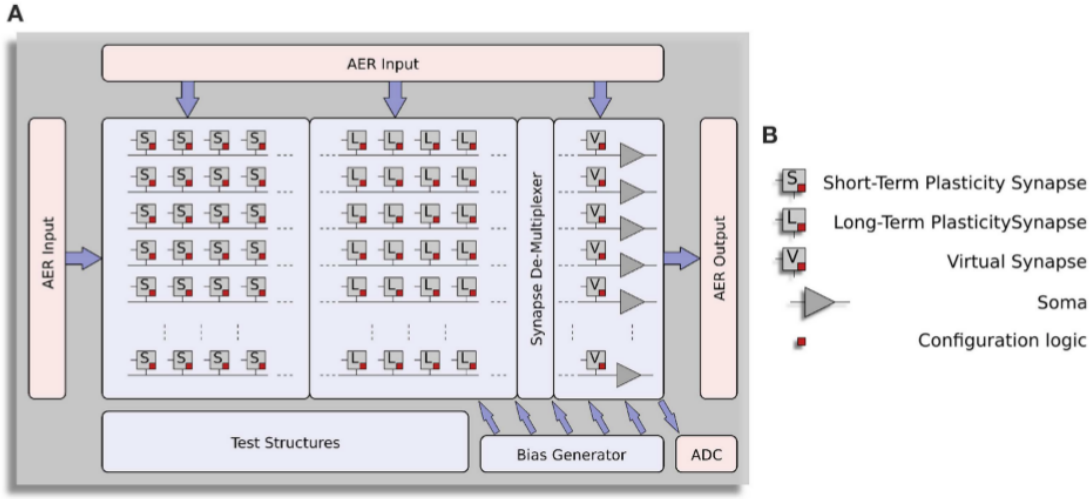

FIGURE 2.1: Block diagram of the chip architecture.

Brader et al., (2007), which reproduces many of observed biological behaviors. A real neuron in the brain receives hundreds of spikes every second and needs to figure out somehow, which signals it should forward. This process is a problem of learning, which neuroscientists know as 'wiring' between neurons. In the last 20 years, a new theory arose, namely Spike-Time Dependent Plasticity (STDP), which suggest, that the temporal distance between the spikes of two neurons is crucial for their wiring (Markram et al., 2012). Although there were many models of STDP proposed, today the theory is suspected not to cover all observed neurological learning behaviors. Therefore, the update rule proposed by Brader et al. was implemented on the ROLLS chip. The synaptic weights are updated according to the timing of the presynaptic spike, the state of the postsynaptic neuron's membrane potential and its recent spiking activity. The following equations describe, how the weight of a given synapse i is updated depending on different thresholds This update rule is evaluated at the arrival of every presynaptic spike.

$$\begin{cases} w_i = w_i + \Delta w^+ & \text{if } V_{mem}(t_{pre}) > \theta_{mem} \text{ and} \\ & \theta_1 < Ca(t_{pre}) < \theta_3 \\ w_i = w_i - \Delta w^- & \text{if } V_{mem}(t_{pre}) < \theta_{mem} \text{ and} \\ & \theta_1 < Ca(t_{pre}) < \theta_2 \end{cases}$$

w_i represents the synaptic weight, $\triangle$w+ and $\triangle$w- are the amplitude of potentiation or depression of the weight, which is triggered by a presynaptic spike. These

two values can be set by the user and can be used to tune the learning behavior. $V_{mem}(t_{pre})$ indicates the membrane potential of the postsynaptic neuron at the time when the presynaptic spike arrives. $Ca(t_{pre})$ is the calcium concentration in the postsynaptic neuron, which is proportional to the recent spiking activity. θ_{mem}, θ_1, θ_2 and θ_3 represent tunable thresholds which can also be used to set the learning behavior.

The amounts of synapses in this array corresponds to a connection from every neuron to every neuron. There are three different modes, in which a synapse can be activated. In the direct activation mode, a synapse is activated by an AER event with the corresponding row and column address, meaning there is a direct signal from one neuron to another. Then there is the broadcast activation mode where a whole column of synapses is activated in parallel and as third option, there is the recurrent activation mode, in which the synapse column of the postsynaptic neuron is activated again after this neuron has spiked.

The third block of neuron/synapse-circuits it the nonplastic synapse array, which comprises also 256 x 256 synapse circuits. This allows a similar connection mapping as the plastic synapse array, but rather than changing the synaptic weights with a on-chip learning algorithm, one can program them. Each synapse has a 2-bit latch to set the weight and an additional latch to set, whether the synapse should be excitatory or inhibitory. In excitatory mode, one can also activate Short-Term Depression dynamics, where the Excitatory Post-Synaptic Current (EPSC) decreases with the number of input spikes and slowly recovers if no inputs are incoming. Like in the plastic synapse array, the nonplastic synapses have also the same three activation modes, direct, broadcast and recurrent.

The communication from and to the ROLLS chip, as well as the on-chip communication between neurons is encoded in Adress-Event Representation (AER). This is an asynchronous handshaking protocol for transmitting signal to, off or between neuromorphic systems. On the ROLLS chip, AE's are used for setting all synapse parameters as well as transmitting spikes on and off the chip. Although neurons work fully parallel, the spike output AE's are serialized as they can only access one shared bus. Possible spike collisions are solved by an arbiter circuit which grants bus access to only one neuron at a time. Other structures on the chip are the virtual synapses, a 256 x 2 array of linear integrator filters, which allow the user to send programmed spikes to the chip and an Analog to Digital Converter (ADC), which converts subthreshold currents from synapse and neuron circuits to digital signals using a linear pulse-frequency-modulation scheme. Summarized is the ROLLS processor a device which allows the user to model

neuronal architectures by choosing neurons and synapses and tuning their behavior by setting different biases. These architectures can be tested by stimulating the neuron circuits from outside and can be reviewed by examining spikes which are obtained from the processor.

2.1.2 Parallella board

The Parallella board (*Parallella* 2018) is a high performance computing platform, which is with a power consumption of 5 Watt also very energy efficient, compared to other platforms. It comprises an ARM-based Dual-core processor and a 16-core Epiphany co-processor with a reduced instruction set computing system-on-chip (RISC SOC). The connection between the ROLLS processors and the Parallella board is interfaced with a FPGA-board, which only receives and transmits the AER events between them. Over AXI-buses, the AER events are transmitted between the ROLLS and the FPGA board as well as between the FPGA and the dual-core processor on Parallella. The 16-core Epiphany processor is not used in this setup.

2.1.3 DVS camera

When working with neuromorphic platforms, using "normal", frame-based cameras has significant drawbacks (Lichtsteiner et al., 2006). If a system requires detection of high-frequency or short-latency vision inputs, a frame-based camera is producing a large amount of data, because it stores redundant spatial or temporal information for every frame, even if nothing has changed since the last frame. The amount of data can easily reach 1 GB/s or more and would be too large to be evaluated and used for the neuronal system in real-time. Another drawback is in the nature of data transmission of sensory information, because in conventional sensory systems, the sensor data gets polled in a small enough time-frame to detect frequencies of interest. But neuromorphic hardware uses the AER protocol, in which the sensor decides when it sends information, so a frame-based camera would have to constantly send new information.
Because of these reasons, a Dynamic Vision Sensor (DVS) camera is often used for neuromorphic systems. It emulates the retina of mammals, allowing event-based vision. This means, it transmits asynchronous and only non-redundant information in form of events (Liu and Delbruck, 2010). The events in this DVS camera are vectors with four components. The first two store the x- and y-Coordinate of

the pixel, which sent the event. The third stores a time stamp and the last gives information about the polarity. The polarity is either "on", if the luminance of the pixel increased or "off" if it has decreased. An event is therefore only created and sent, if the polarity of a pixel has changed (Milde et al., 2017).
This leads to a rather small amount of data which gets transmitted and therefore to a smaller power consumption. Also allows the DVS camera a higher processing speed, as the transmission latency is reduced.

2.1.4 Pushbot & eDVS

The Pushbot is a small (10cm x 10cm chassis), autonomous robot used in this work. It has two motors driving the left and right propulsions individually. Each motor can be set to hundred different speed levels, being as fast as an average walking person on highest speed (*Pushbot* 2015). The main piece of the robot is an embedded DVS board (eDVS) (*DVS* 2015), which is composed of a DVS chip connected to a 64MHz ARM7 microcontroller. Over this controller, the motor commands are set. The eDVS contains additionally a Inertial Measurement Unit (IMU) which provides the current orientation. However in this project, the IMU was not used as it was found not to be reliable. The Pushbot further provides WLAN connectivity up to 12 Mbit/s which allows to set the velocity and read sensory data remotely with a latency lower than 10ms (Milde et al., 2017). The WLAN connection was used in this work, as the robot is too small to carry the ROLLS processor and Parallella.

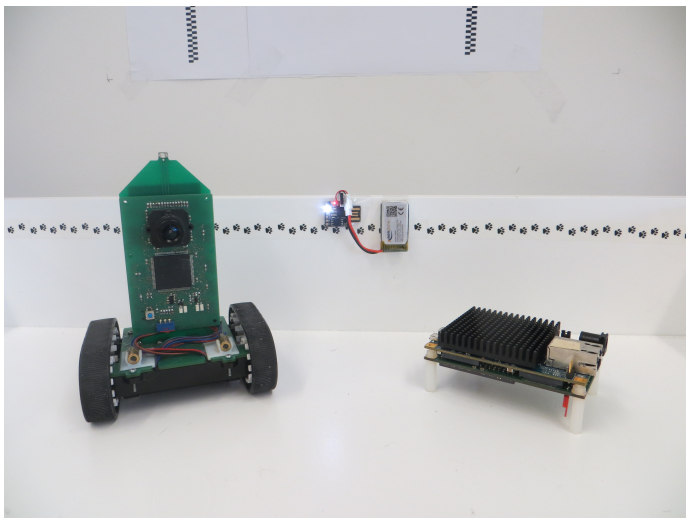

FIGURE 2.2: Left is the Pushbot with a mounted eDVS. On the right is the Parallella board and ROLLS chip. In the middle is one of the blinking LED's used in this work.

2.1.5 Setup

This section covers shortly the overall setup. The Parallella board is connected with the ROLLS processor over an FPGA as already described in the section 2.1.2. The Parallella, as well as the Pushbot, gets connected over WLAN to a dedicated network. Over this network, the user can upload code to the Parallella and send commands over a Secure Shell (SSH).

2.2 Software architecture

Software written in C++ is used in this work to program the neuronal network, to evaluate and react to data from the robot and the ROLLS, and to program the behavior of the robot. It can be divided into three parts, a helper library, the controller class and the main class. The helper library consists of many functionalities for using the hardware pieces like the ROLLS oder Pushbot. For this project, the most important helper classes were probably the robot and the ROLLS device class for instantiating and controlling the robot and ROLLS, and the ROLLS architecture class for programming the neuronal network. In the main class, a robot and a ROLLS object are instantiated and a connection is set up with the robot. Then some listener objects are instantiated, which can catch events from the eDVS, ROLLS or keyboard. Finally a controller object is instantiated and the robot and the ROLLS object are passed to it as arguments. The controller class is the core of the software architecture, which connects the different pieces. Within it, the user can create a neuronal architecture and set for every listener, how incoming events should be processed. This gives the opportunity to react to spikes from the ROLLS, to eDVS events or even to key inputs, which is very useful for steering the robot or stimulating neurons on the ROLLS from outside.

2.2.1 Visual frequencies detection

To detect the blinking lights, which are used as landmarks for the experiments in this work, a piece of code was adapted originally written by Julien Martel at INI Zurich. It stores every incoming eDVS event in an two-dimensional array consisting of lists, one for each pixel. Then the time stamp of the current event gets compared to the time stamp of previous events from that pixel to check, if the difference matches the blinking frequency specified by the user. This comparison is not only done with previous events from the current events 'own' pixel but with

every surrounding one. The older events are classified as matching ones, if the timespan between the compared events lies in the border of the frequency period plus some variance, and as distracting events, if it does not. The current event does only count as a blinking light event, if the number of matching neighboring events is high enough and the number of distracting events does not exceed some value. These minimum, respectively maximum values for matching and distracting events are also set by the user. Together with the period and variance parameters, one can tune how coarse or fine the frequencies are differentiated and detected. When using multiple frequencies like in this work, the events are sometimes classified as the wrong frequency. Because of that, the corresponding LED populations are not stimulated, until a certain threshold of events classified as a certain frequency is reached. As the vast majority of blinking events are associated with the right frequency, this ensures a reliable detection-and-fire mechanism. In this project, only the events from a 10 x 10 pixel window in the middle of the DVS were evaluated so that the lights are only detected if the robot is really facing their direction.

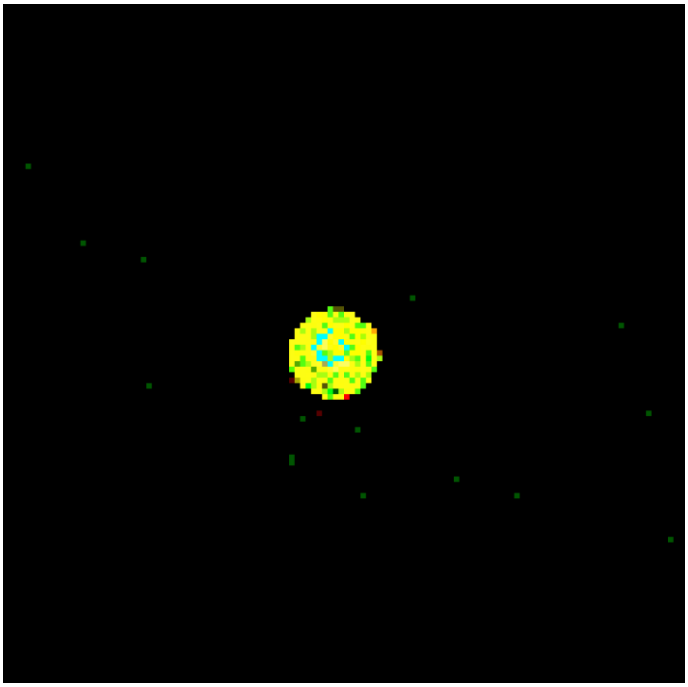

FIGURE 2.3: A snapshot of the 128 x 128 pixel eDVS pointed to a LED light.

2.3 Neuronal Network Architecture

It has been shown, that many insects and rodents can orient themselves, amongst other mechanisms, with help of Head Direction (HD) cells (Seelig and Jayaraman, 2015). In rats, these HD cells were found in the dorsal presubiculum, a region in the brain located in the hippocampal formation, which is located again in the temporal lobe and in the limbic thalamus (Zhang, 1996). The HD cells work as an orientation network in the horizontal plane, as the view between 0°and 360° is equally distributed between the cells. A HD cell fires, if the rat is headed to the preferred direction of the cell. If the rat is changing its direction, the firing activity is shifted to other HD cells and therefore the inertial perception of direction is updated, even in darkness (Song and Wang, 2005). This suggests an update mechanism relying on self-motion. However, also landmarks, meaning orientation points in the surroundings, play an important role in the orientation mechanism. The HD cells are also coupled to these landmarks, such that if the landmarks would be rotated, the whole head direction system is rotated too. It has been shown that the landmark information is preferred, if there is a conflict between the inertial orientation and the known landmarks, meaning that landmarks can reset the inertial orientation. If placed in a new environment, the HD cells of the rat can therefore quickly adopt a new preferred direction relative to new landmarks (Zhang, 1996).

To model such a HD network observed in animals, different approaches have been taken in the past. One example is a mapping between the angular velocity and head direction resulting in a jointly neuron population which can predict the future head direction. Another one suggests to model the HD network as a circular shift register with attractor dynamics, meaning that the network activity stabilizes itself always to certain cells, also called the hill of activity. In this model the HD cells are connected with left and right rotation cells, which in turn are activated by vestibular cells. The rotation cells then activate neighboring HD cells of the ones that currently make the activity hill (Redish et al., 1996).

In this work, a similar abstraction of that model was chosen, based on the work of Kreiser et al., (2018). The HD cells are modeled by a ring attractor network consisting of 36 Neurons. The network implements a soft Winner-take-all (WTA) dynamics, meaning that the activation of neurons converges always to only two or three neurons forming the hill of activity. To sustain the activity, the neurons are recurrent. They have a local excitatory connection with themselves and their direct neighbors and a globally inhibitory connection with every other neuron.

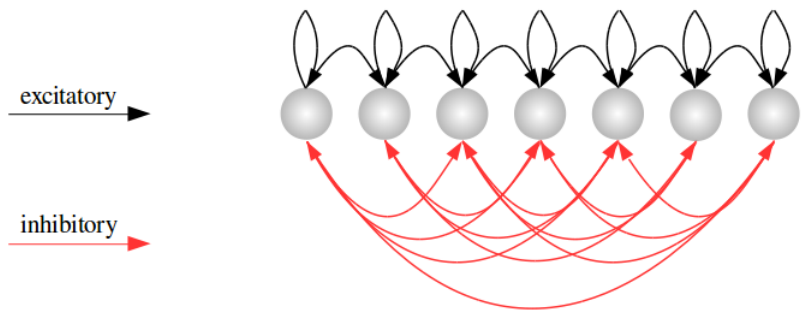

FIGURE 2.4: The arrows indicate excitatory and inhibitory connections. The activity converges always to a few neurons.

To shift the activity in the HD neuron population, there are two shifting populations, for clockwise and counterclockwise shifting, each containing also 36 neurons. These direction terms are chosen deliberately over "left" and "right", because the HD population as well as the shifting populations are circular.
Between the shifting populations and the HD population is another 36 neuron population, which integrates the head direction. This means, the shift populations have a feed forward connection to the Integrated Head Direction (IHD) population, which is shifted by two neurons either clockwise or counterclockwise. The IHD population shifts the activity of the HD population then by a direct feed forward connection. In this modeling, shifting the activity is initiated by moving the robot. Or more precisely by the same key command as for starting the motors of the robot, because this stimulating input is, as well as the motor command, sent over WLAN and therefore this setting is reducing latency compared to an activation over the robots wheel encoders. The shift activation is realized by two drive populations each counting five neurons, which activate the corresponding shift population through an all-to-all connectivity. In order to activate the correct IHD neurons, the shift populations get cross-inhibited by the HD population, meaning only the corresponding shift neurons to the active HD neurons are not inhibited and can therefore be active. In this work, the shifting speed is not depending on the angular velocity of the robot, as the robot is always turning with the same velocity.

So far, the described setup allows a sustained hill of activity and its shifting. As the goal of this thesis is to implement and test the use of visual landmarks to correct the activity of the HD population, there has to be a mechanism for learning the positions of these landmarks. To enable this, there are neuron populations of

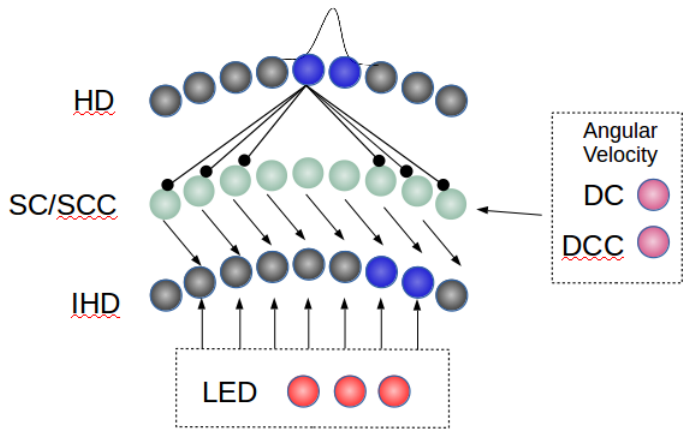

FIGURE 2.5: The network architecture with the schematic connections between the different neuron populations. Head Direction (HD), Integrated Head Direction (IHD), Shift Clockwise (SC), Drive Clockwise(DC), Drive CounterClockwise (DCC) and the LED populations.

three neurons for every LED light in use. If a light is recognized, the corresponding population gets stimulated. The HD neurons and the LED populations are connected over plastic synapses, which allows to learn the positions of the LED lights relative to the head direction.

2.3.1 Biases

The ROLLS processor offers for the different circuits like neurons, plastic and nonplastic synapses different tunable parameters which are set by the bias generator. As the circuits emulate biological properties of neuronal cells, one can use the biases to change the behavior of the circuits with respect to these properties. The neuron circuit has for example the two thresholds IF_TAU and IF_RFR which model the leak time of a neuron and the refractory period. With a higher leak time, the action potential of the neuron gets shorter and with a higher refractory period, the neuron takes more time until it can fire again. These are only two examples of a wide range of biases for every circuit and for this project, it was crucial to fine-tune them, although some values could be taken from earlier projects. The most important biases for this work were on one hand the excitatory and inhibitory weights of the nonplastic synapses. For the whole neuronal architecture to work, the different populations had to have different weights in order to activate some populations strong enough and not inhibit others too strong. On

the other hand, the biases of the plastic synapses had to be tuned right to enable learning between the HD and LED populations. In the following table, the important biases for this work are listed. The SL_THMIN bias changes, if the keys for learning and stop learning are pressed. How this mechanism works and why the bias values were chosen like this is explained in the results chapter in section 3.1.2.

name	*description*	*value*
NPA_WEIGHT_EXC	Excitatory weight 1 of nonplastic synapses	456 pA
NPA_WEIGHT_EXC0	Excitatory weight 2 of nonplastic synapses	36.5 nA
NPA_WEIGHT_EXC1	Excitatory weight 3 of nonplastic synapses	19.3 µA
NPA_WEIGHT_INH	Inhibitory Weight 1 of nonplastic synapses	659 pA
NPA_WEIGHT_INH0	Inhibitory Weight 2 of nonplastic synapses	969 pA
NPA_WEIGHT_INH1	Inhibitory Weight 3 of nonplastic synapses	1.58 µA
PA_DELTAUP	Controls the amplitude of a digital up jump	22.1 nA
PA_DELTADN	Controls the amplitude of a digital down jump	363 pA
PDPI_TAU	Controls the time constant of plastic synapses	4 pA
PDPI_THR	Controls the threshold of plastic synapses	2.55 nA
PA_WHIDN	Sets the strength of the plastic weight gain	24 µA
SL_MEMTHR	Controls the membrane threshold	105 pA
SL_THDN	Controls the digital up/down window	6 pA
SL_THMIN	Lower bound of the digital up/down window	4 pA

TABLE 2.1: chip biases

3. Results

In this section, the outcome of the work and the results of tests are described, but also the process of reaching this and some intermediate results are shown.

3.1 Neural Network

3.1.1 Intermediate Results & Development

The first step was to reproduce the architecture of a spiking head direction neural network, which has been implemented by Kreiser et al., (2018). For this, I had to get acquainted with the hardware pieces and overall software structure which connects them. After I immersed to the subject and worked trough the code, I got a better understanding of how the setup works and could start implementing the neural network.
At first, I made a somewhat 'naive' approach, where just all neuron populations were instantiated and connected to each other like it was described in the paper. Then a little robot steering mechanism over the keyboard was implemented, which also activated the drive populations on the ROLLS chip. Though the robot was turning and the drive populations were activated, the neural network behaved not like expected. There was no or not a continuous shift of the activity in the Head Direction (HD) population. I realized, that the weight of the connections and the on-chip biases of the neurons and synapses play a important role in the functioning of the network. With some help, the biases were tuned right and the connection weights between the different neuron populations were adjusted. Also the soft Winner-take-all (WTA) functionality of the HD population was changed to a hard WTA. This means, that the hill of activity did not consist of three neighboring neurons anymore but only of one single neuron. This made it easier for the Integrated Heading (IH) population to actually set the activity to a new neuron in the HD group. After these corrections and adjustments, the

network showed the desired behavior. Meaning, that the activity in the HD population shifted, when the key which let the robot turn, was pressed. Figure 3.1 shows a trial with this network architecture.

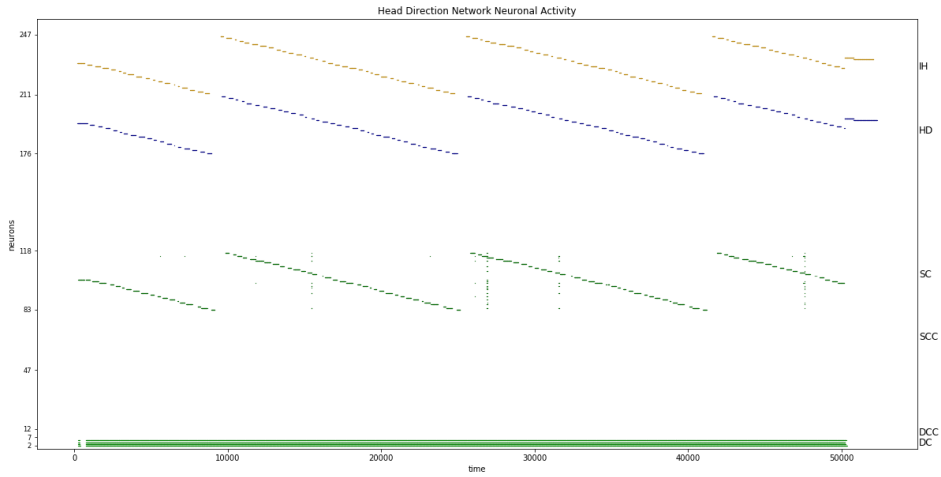

FIGURE 3.1: The neuron activity during three turns counterclockwise. The labels on the right side are from bottom to top: 'Drive clockwise', 'Drive Counterclockwise', 'Shift counterclockwise','Shift clockwise', 'Head Direction', 'Integrated Head Direction'

After this first success, unfortunately the ROLLS chip stopped working because of a broken connection between the chip and the Parallella board. The second ROLLS chip which was further used, did not fit with the current way, the neural network was implemented and the biases were set, due to mismatch in the circuits.

The first problem was, that the neurons of the HD population did not have a high enough self-excitation and could therefore not sustain a stationary activity, which is of course needed when the robot does not move. To recall, the HD group was currently implemented with a hard WTA, so only one neuron should have been active at a time. To ensure that the activity does not fade, one could have raised the strength of self-excitation of the HD neurons. But if that is too high, an activity shift would not longer be possible because a higher self-excitation also leads to a stronger inhibition of all other HD neurons and the IH neurons could not stimulate other HD neurons enough to let the hill of activity shift. Therefore another approach had to be taken to stabilize the activity of the HD neurons, namely changing back to a soft WTA network. This had the effect that now three neurons were active at a time and stimulated each other, which led to the desired stability. The drawback of this soft WTA network is, that the head direction information gets less precise, as one neuron covers 10°of the view and if three neurons are

active at the same time, this hill of activity covers a wider range of 30°. Although one could argue that if the shift still changes at one neuron at a time, then the precision does not decrease because only the neuron in the middle of the activity hill represents the head direction.

Although there should be three HD neurons active at the same time, in reality often only two are spiking. This is probably due to the fact that the different neuron and synapse circuits on the chip do not have the same output current even if the have the same input currents and same biases. If the two neurons left and right of the activity hill are not equally strong, one will eventually inhibit the other stronger and that one will stop spiking.
The next problem was, that the activity hill of the HD populations did not move, when the drive population were active and the robot turned. Although the drive populations excited the correct shift neurons and these again activated the shifted IH neurons, it seemed that the IH neurons did not have any influence on the HD neurons and the activity hill remained in the same position. To tackle this, the shift was changed from one to two neurons, meaning that one shift neuron stimulates two IH neurons, like shown in figure 3.2. This leads to one more neighboring HD neuron which gets stimulated by the IH group. After this adjustment, the network showed some shift activities, but still the activity hill got stuck eventually at some point in the HD population and didn't shift further.

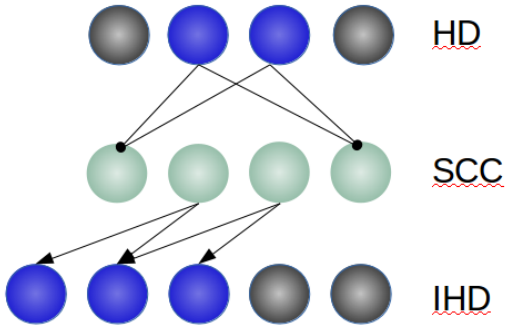

FIGURE 3.2: A diagram of the shifting mechanisms with two activated IH neurons per shift neuron.

Now there would have been the possibility to change the chip biases, for example the weight of the synapses, to change the behavior of the network. As the shift is ultimately initiated by the drive populations, a possibility would be to increase the excitation weights of the synapses between the drive and shift, between shift and IH and between IH and HD population, as the drive neurons feed forward to the shift, the shift to IH and IH to HD neurons. But these weight were already

on the highest possible setting, there was no room for improvement at this front. Another possibility would have been to decrease the self-excitation of the HD neurons, which would allow another HD neuron to inhibit the active one more easier. For the same reason, one could also lower the inhibitory weight. Implementing that, the self-excitation was decreased to the point were the HD neurons still could sustain an activity hill and the inhibitory weight was carefully chosen in a range where it was high enough to ensure the functionality of a WTA network, meaning that the spiking activity converges to only two or three neurons, but not high enough that no more shift was possible because the HD neurons inhibit each other to much.

After fine-tuning these synapse weights, the shift mechanism appeared to work better, but the problem of the activity hill which gets stuck, still remained at some neurons. As the potential for improvement by changing the chip biases seemed to be exhausted, the only possibility to overcome the problem of different neuron strengths due to device mismatch, was swapping the neurons of the different populations, but especially the HD group. At this time, the software structure was written in a way, which restricted the neuron groups to being continuous. Therefore to change the neurons in a population, the whole neuron group had to be moved. As there are only 256 neurons on the ROLLS chip, there was only a rather small scope for the replacement of the populations. After moving the HD and IH group, it became obvious that the problem of too different neurons was existing on every region of the neuron array. As a consequence, the software architecture of the neuron groups was rewritten to allow a group to contain any, spatially independent neurons. With this feature, one can choose specific neurons and drop the ones which seem too strong or weak.

Now it made sense, to test the behavior of the network thoroughly and to spot the responsible neuron, when the activity hill got stuck. This was a rather difficult and time-consuming process, as it is no clear if the current HD neuron is too strong or the neighboring is too weak or even there is a weak neuron in a shift or the IH population which prevents the shifting. So first, all the populations were tested for not responding or too weak neurons (3.3), which where then replaced.

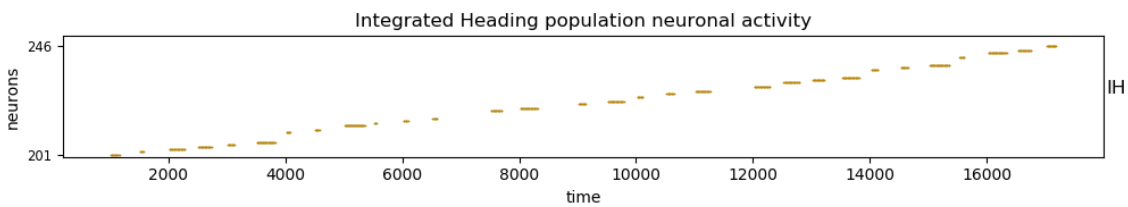

FIGURE 3.3: A test of the IH population. Neurons, which did not spike long enough or at all were replaced.

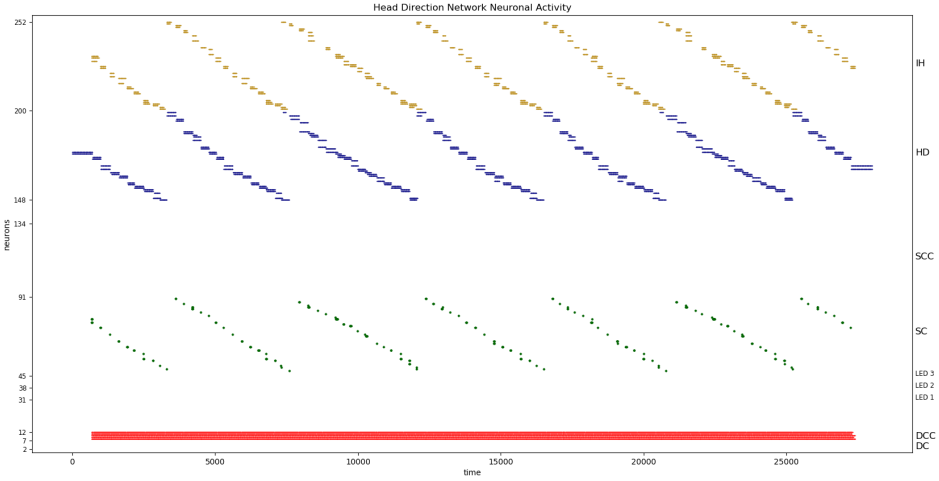

FIGURE 3.4

After it was ensured, that every population contains only working neurons, the ones of the HD and IH group at the critical locations where the activity still got stuck, were replaced until the shifting went smoothly trough the HD population.

Figure 3.4 shows the neural activity of the network while the robot is doing six turns clockwise. The neurons of the HD (blue) and the IH (yellow) do not fire equally long in every turn, which is a consequence of the device mismatch or different neuron strength. This leads to a random drift in the configuration of the HD population relative to the head direction in every turn,which can accumulate over multiple turns. Figure 3.5 depicts the mean and standard deviation of the relative and absolute neuron drift of 10 trials in both turning directions. The relative drift is the distance between the neurons when the robot made one turn, the absolute drift is the distance from the neurons which were active at the start and the neurons which were active after all turns. As the figure shows, the mean error fluctuates around zero for every turn, which indicates that the neuron drift is not caused by bad calibration of the robot's motor speed relative to the activity hill propagation, but only from random drifts due to neuron mismatch. The standard deviation lies between 20° and 30° for the relative drift and between 20° and 55° for absolute drift.

The next step towards the goal of the thesis was adding neuron populations, which respond to the blinking LED's and connect them with the HD population to enable learning between these groups. First the code which can detect blinking frequencies from incoming eDVS events was configured with respect to the chosen lights and their frequencies and was tested. After ensuring that the software

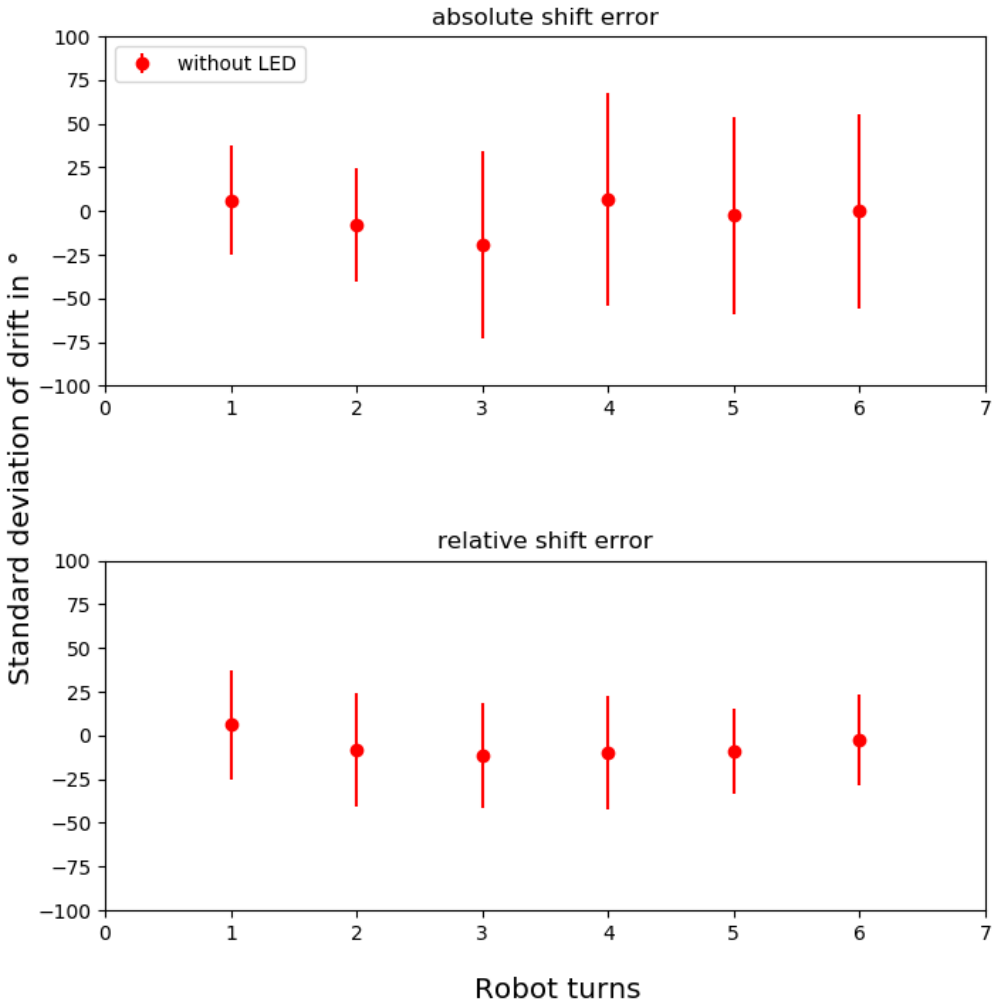

FIGURE 3.5: Mean and standard deviation of relative and absolute shift error during 6 robot turns.

can detect them individually, there was set a mechanism which activates the LED neuron population when the corresponding light was detected.

3.1.2 Learning

In order to enable learning between the LED and the HD group, the plastic synapse biases have to be set right. As already addressed in the methods section of this report, the ROLLS chip implements a weight update rule, which can be tuned over multiple biases. These biases, such as the amplitude of the synaptic weight or the membrane threshold determine for example how fast a synapse weight gets potentiated or depressed. Usually, a synapse weight gets potentiated, if the

pre- and postsynaptic neuron fire successively multiple times and gets depressed if that is not the case. As seen in figure 3.1, the neuron drift in the HD population will often already take place after one robot turn. If a LED is placed somewhere around the robot, the chance will be high, that not the same HD neurons are firing when the light is seen the first and the second time. Therefore, the learning has to take place immediately when the light is recognized and the LED population fires for the first time. The consequence of such an learning behavior is, that learning will always take place if an LED population fires, which is not intended as the LED lights should later be used as correctional landmarks. To tackle this problem, a mechanism has been implemented to toggle the learning on and off over keystrokes, by changing the biases of the chip. If learning is enabled, the bias SL_THMIN is set at the lowest value possible. The other biases are set in a way, that learning always takes place, if there is a pre- and postsynaptic spike at two connected neurons. When learning is toggled off, the SL_THMIN is set to the highest possible value, absolutely preventing any learning. This enables to turn learning off after all LED lights have been recognized for the first time.

3.1.3 Path Integration & Mapping

To see, if the connection over the plastic synapses works, different tests have been conducted with both connection directions, from the HD to LED group and otherwise. For this, the robot was placed in an arena, where the LED lights were fixed on the walls. The first experiment was about whether the network is able to form a map in terms of the 360 degree view of the robot, meaning if the network can learn and remember the position of the LED lights. For this, the neuron groups were connected from HD to LED, so that the presynaptic spike comes from the HD group and the postsynaptic spike from the LED group. The robot turned only once, then all the HD neurons were stimulated one after another to check, if some synaptic connections were potentiated. After some trials, it showed not to be working. The problem was, that as soon as the LED neurons spiked, the plastic synapse between the HD neurons active at the moment and this LED neurons was potentiated and the LED neurons kept firing. When the activity hill shifted to the next HD neurons, the LED neurons were still firing and also the plastic synapses between the new HD neurons and the LED neurons became potentiated and so forth. This means although the robot turned away from the light, the LED neuron kept firing and made connections to more HD neurons as shown in figure 3.6.

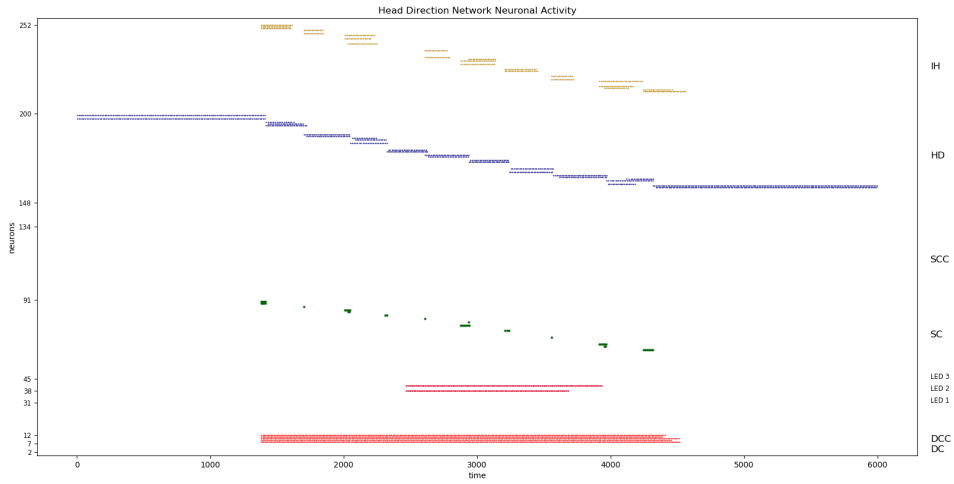

FIGURE 3.6: Trial with connections from IH to LED neurons. Multiple connections are formed as the activity propagates.

As a countermeasure, the LED populations were set to inhibit themselves, which should prevent them to keep firing after receiving input spikes. It seemed that this was working because the LED neurons actually fired only for a very short period, but now another problem appeared. Although all lights are recognized and the corresponding neuron populations were activated, no LED neuron fired after the robot turn, when all HD neurons were stimulated. Apparently, the amount of time the LED neurons where spiking was too short for the learning to take place. To tackle this problem, the HD population was disconnected and the IH neurons were connected to the LED population. After the self-inhibition of the LED neurons was removed and they spiked longer again, the mapping of the LED lights worked.

Figure 3.7 shows a trial of the robot doing one turn. Before the turn, learning was activated and after standing still again, deactivated. Then, all the neurons of the IH group were stimulated to check if some learning took place. For this trial, two lights were blinking and as shown in the figure, they spike as a reaction to the same IH neurons, which were active when the system detected the lights during the turn of the robot. This shows, that the system is indeed able to form a map of its surrounding landmarks, although its not very precise because the spike train of the LED neurons is rather long.
To test, if the network can learn the position of the lights and correct its path integration with help of these visual landmarks, the LED populations were connected over the plastic synapses to the HD neurons, meaning in the other direction as in the mapping experiment. The setup of this test is the same as before, only that

the robot is doing more than one turn. Learning is only enabled in the first turn, in the following turns, the network then should be able to correct shift errors by resetting the hill of activity in the HD neurons through the connection from the LED populations. It turned out that in this setting too, it makes more sense to connect the LED neurons with the IH group. The problem was again the device mismatch, meaning that the activity is only corrected if the learning between LED and HD took place with one of the stronger HD neurons, because only these were able to shift the activity in their WTA-network.

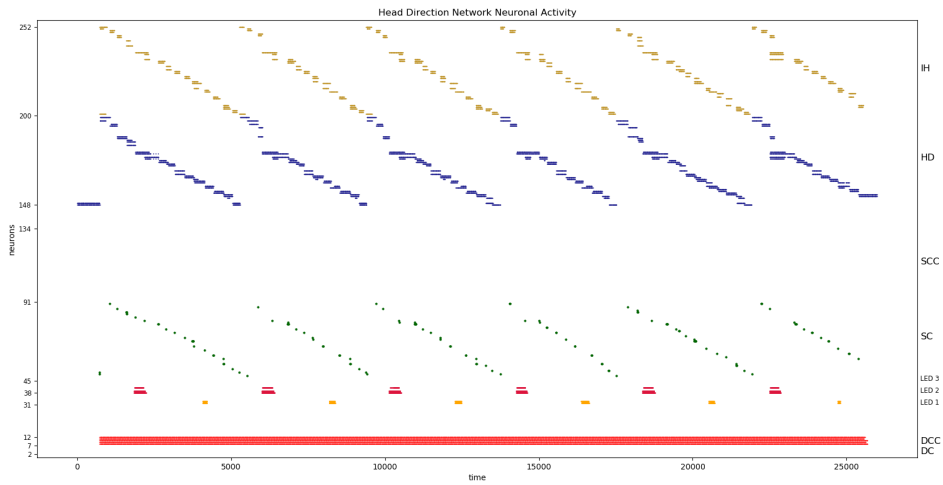

FIGURE 3.8: A trial over 6 turns clockwise with two LED lights.

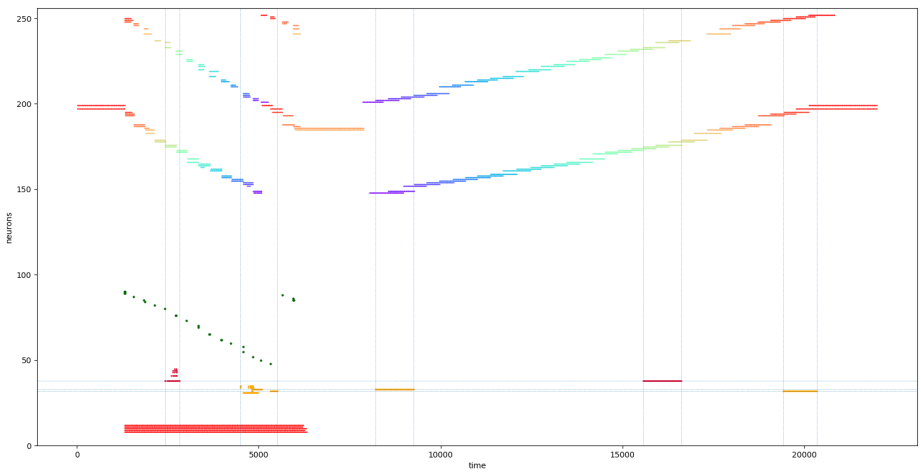

FIGURE 3.7: During one turn of the robot, the positions of the LED lights relative to the internal horizontal orientation of the neural network. Then the Integrated Heading neurons were stimulated one after one.

With the connection between LED and IH populations, the network showed that learning of the landmarks and correction of the shift error is possible. The network was tested with one, two and three lights mounted on different walls of the arena, like figure 3.9 shows. For each setting, 10 trials with the robot turning left and 10 trials turning right were carried out. During one trial, the robot made six turns and was stopped at the starting position. Figure 3.8 shows an example of such a trial. One could argue that the measurements can be falsified because the robot is stopped by hand and does therefore probably not always stop after exact six turns, but as the network can differentiate only between 10°of the view (360°/ 36 HD neurons), this method should be precise enough.

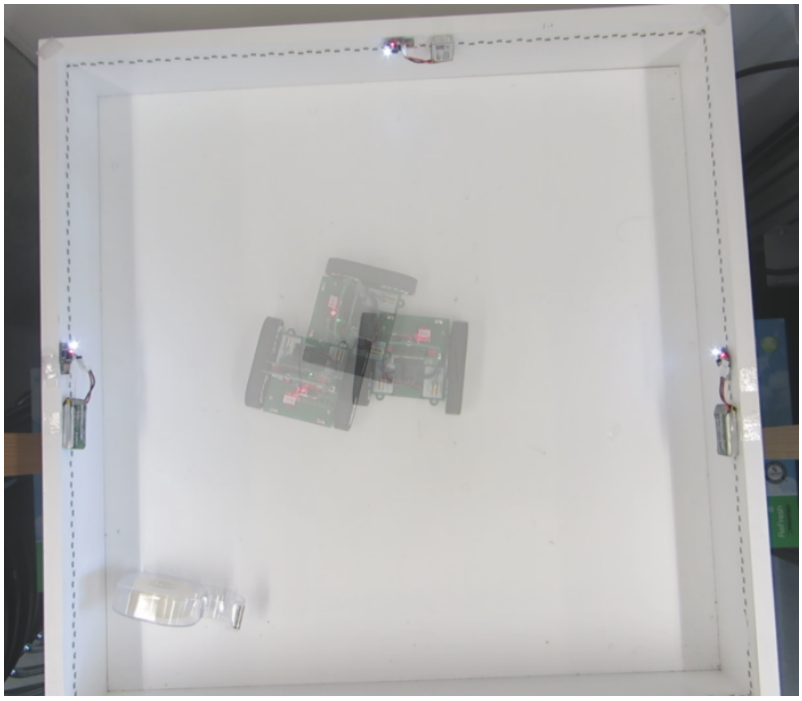

FIGURE 3.9: A graphical demonstration of a trial. The LED's are mounted on the left, back and right walls.

The data was evaluated as follows: The timespan between the first and the last spike of the drive populations was divided by the number of turns, then the time stamp after every turn was calculated. Thus the spiking neurons after every turn could be determined and compared.

Figure 3.10 shows the mean and standard deviation of the relative and absolute shift error for the different light numbers and the trials without lights for comparison. Again, the absolute shift error is the difference between the starting neuron and the actual neuron, the relative shift error measures the difference between the active neuron one turn before and the active neuron now. Because the position of the lights is learned in the first turn, the absolute shift error is set to zero after the

first turn and measured from there because the error in the first turn cannot be corrected and says therefore nothing about the reliability of the correction mechanism.
Figures 3.10 show that the means of all trials fluctuate around 0°. This means that the error can not be caused by a false setting of the robots motor speed, because if that would be the case, the error should always be positive (too fast) or negative (too slow) and the means would drift in a certain direction. The standard deviation of the relative shift error remains for the different light numbers used more or less the same.

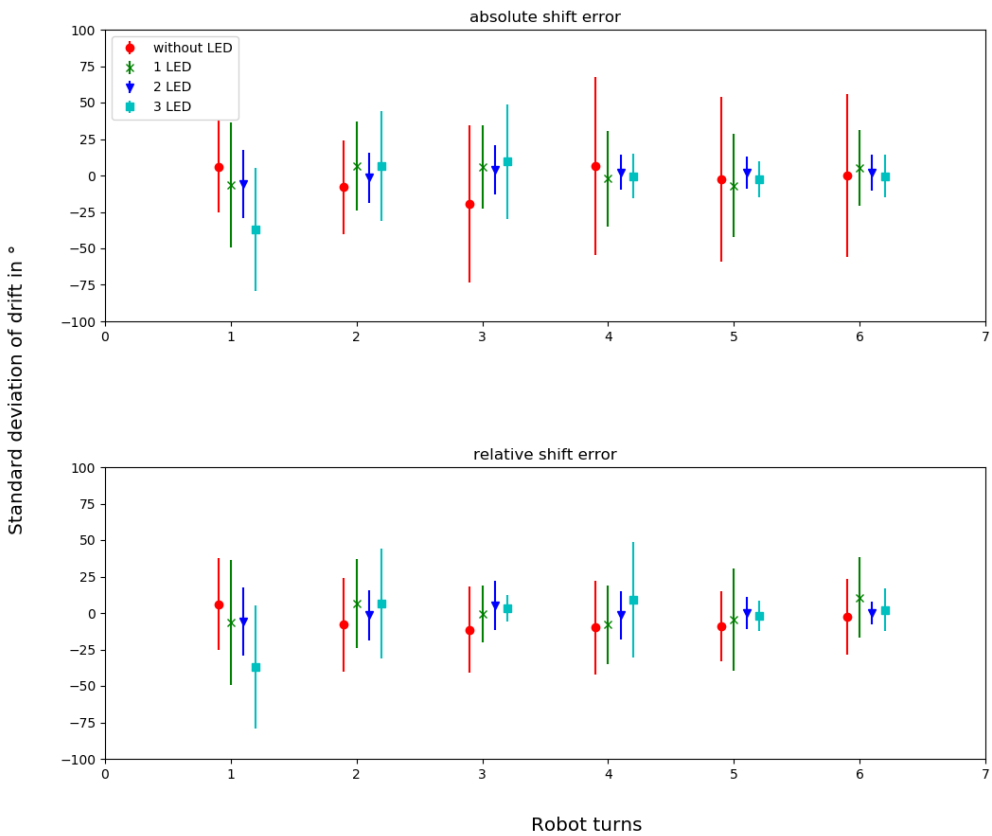

FIGURE 3.10: Relative and absolute shift errors over 6 turns for the amount of LED lights in use from zero to three.

Figure 3.11 **left** shows the average relative standard deviation for each light number over all turns. As the relative error only accounts for every single turn, there seems to be no improvement over one turn when adding one LED light to the environment. However using two lights approximately halves the relative shift error from about 30° to 15° but with 3 LED's the average error spread grows again. When comparing these results with the absolute error, it is remarkable,

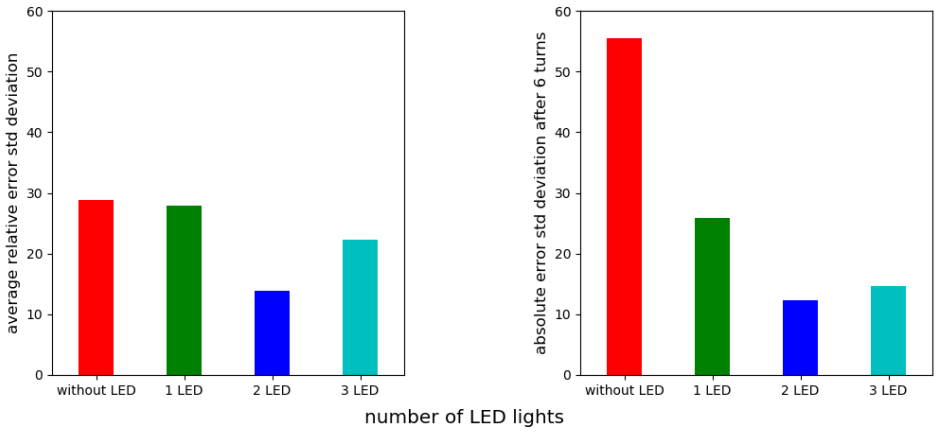

FIGURE 3.11: **Left** Standard deviation of the absolute shift error after six turns of the robot.
Right Average of the standard deviation over all turns of the relative shift error.

that the standard deviation of the shift error becomes smaller already by adding one light. Also it stays within certain boundaries or gets smaller in the trials with some light, while the error spread without any LED landmark grows, as seen in figure 3.10. This shows, that the activity drift can be corrected by the input of the LED neurons and that the learning of the light positions has an influence on the precision of the network.
Figure 3.11 **right** depicts the error spread with different light numbers after six turns of the robot. It gets approximately halved from zero to one to two lights in use, from 55° to 25° to 12°, but then rises again slightly when using three lights. Although these results show some improvement of the network's precision, the question raises, why there is still a rather big spread in the shift error, especially for the relative error in the trials with one and three lights. By having a closer look, it turns out, that in about a quarter of all trials, either the lights are not always detected or something in the learning process went wrong, which means that either no learning took place or there was a connection formed between too many and wrong neurons. This leads to none or false reset of the activity hill and can produce large errors.

Figures 3.12 and 3.13 show the results of only the trials where no disturbing events took place. The average of the relative error's standard deviation over all turns drops significantly for the trials with one and three landmarks, but does not decrease much witch two LED's. The absolute error spread after 6 turns decreases for every number of lights only a little. What this means will be discussed in the next chapter.

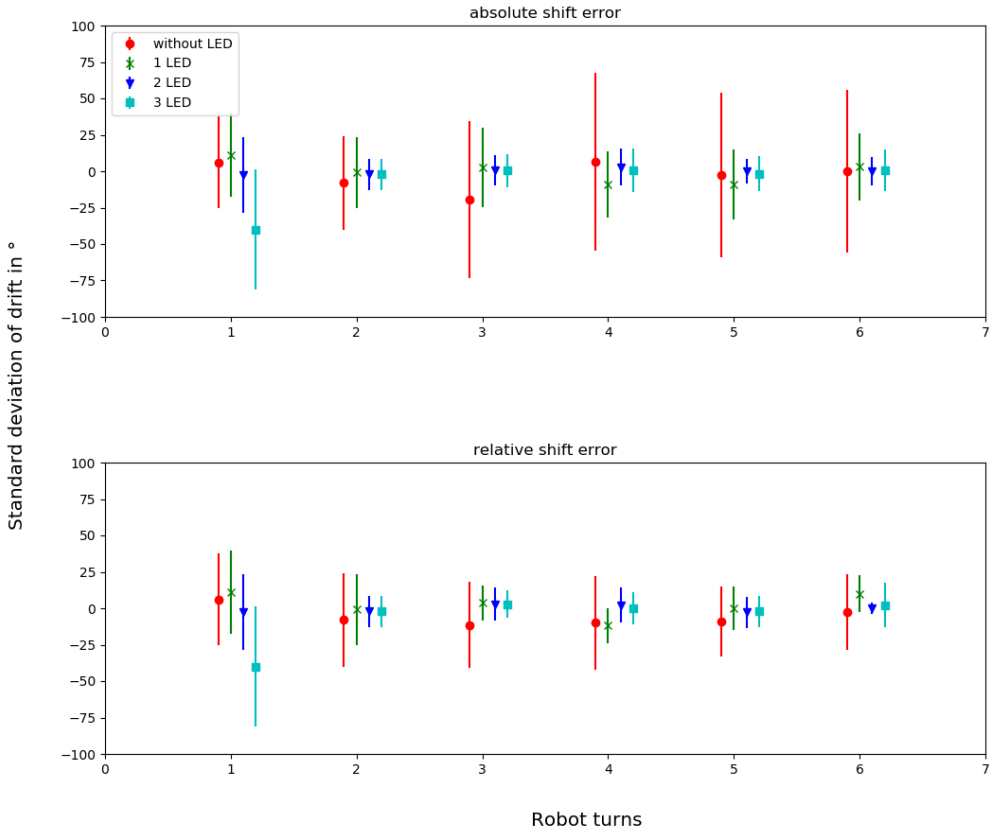

FIGURE 3.12

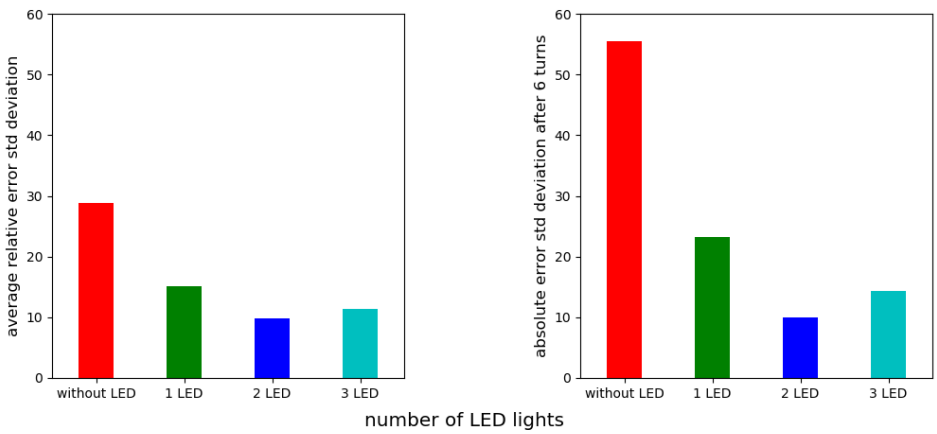

FIGURE 3.13

3.2 Software

During the development of this thesis, several software pieces had to be written or rewritten to create and adjust the neural network. The code is in the appendix of this document, but in this section I will describe the different software pieces I created or changed.

3.2.1 Restructuring

To facilitate the understandability of the software structure, the code was separated into a main and a controller class. The idea behind this is, that future users new to the system don't have to change anything in the main class and can program the behavior of the robot, ROLLS and terminal inputs only in the controller class.

3.2.2 Support of different configuration mechanisms for the ROLLS chip

At the moment, there are two methods to create a neural network architecture on the ROLLS chip. They both use an interface which connects the high user level to the low-level ROLLS address event library. The first method allows the user to create new neuron groups and declare connections between them in the code of the controller class, while the other parses a JSON file, which has all the needed information to create the network. Both of the methods have their advantages and drawbacks, the first method supports many connection types and it is rather easy to add new ones but its not so fast to change things because the controller files have to be recompiled after every change, which takes extraordinary long on the Parallella board. The parsing method on the other hand can adapt to configuration changes very fast because nothing has to be recompiled but at the beginning of this work it supported very few connection types.
When the software code was handed to me, the user had to decide for one of the configuration methods, because the current state of the software did not support using both simultaneously. During this work, the methods were merged, allowing the user to take advantage of both. This was realized with different flags the user can set when starting his experiment on the Parallella board. The program can adapt to either parsing only a JSON file, setting only the synaptic connections declared in the controller file or doing both, depending on which flag is set.

While working on another project with the ROLLS chip, Sebastian Glatz developed a python routine, that created the JSON file to set the synaptic connections. This provided an easy way to add new connection types to the parsing method, which was also used in this project. The merging of the two configurations methods as well as adding new connection types with help of mentioned python routine made it possible to change a neural network architecture fast and easy, which was crucial during this project to tune the head direction network and make it running.

3.2.3 Support of discontinuous neuron populations

Another thing that was very important for this project, was the ability to pick random neurons from the chip and put them together in a group. As already discussed, due to device mismatch, not every neuron circuit has the same properties in terms of conductance and behaves differently when injected with the same current. This leads to unpredictable behaviors of the implemented network and caused also in this work some problems. To avoid this, its helpful when the user can leave out single or multiple neurons, if they seem to disturb the functionality of the network. At the beginning of this project, a neuron population was declared only trough the start and end neuron, forming a continuous list. By introducing the neuron vector, simply a neuron group with a vector as the underlying data structure, the foundation for support of discontinuous neuron populations was laid. Then all functions of the helper library which took a neuron group as a argument, such as synaptic connection functions, had to be rewritten to apply also to neuron vectors. Finally a parsing functionality was added to parse neuron vectors from the JSON file. This allowed fast exchange of neurons from a population.

3.2.4 Neuron testing

In order to obtain information about the strength of the neuron circuits, some testing functionality had to be implemented. A more precise method to do this would be to investigate the neuron circuits of the chip with a oscilloscope to find out about the voltages, but this would have gone beyond the time frame of this work. Therefore different functions were implemented which allow the user to test single neurons or whole neuron groups by stimulating the circuits with spike trains. Information about the strength of the neurons then can be observed by

comparing the spiking activity of the neurons over the web interface or by plotting the neural activity.

3.2.5 Robotic drive and steering mechanisms

For this project, the used robot had to drive either clockwise or counterclockwise, while the corresponding neuron group had to be active. As there was not yet a direct connection between the movement of the robot spiking neural activity on the chip, the specific neurons had to be activated 'manually'. To change this, two threads are created when the program starts, which catch the predefined key events to start or stop the motors of the robot. These threads also stimulate the drive neuron populations, if the corresponding motor is active. To avoid contention, the threads sleep for an appropriate amount of time if no drive event is caught.

3.3 User Guide

When I started this project, it was difficult to get acquainted with the software and it took a long time until I did the first trials with the robot and some neural network. To make it easier for other people, which have to work with this setting for the first time, I wrote a little user guide on getting started with the ROLLS, Pushbot and Parallella board. It is also in the appendix of this thesis.

4. Discussion

In this section, I will reflect the work done in the course of the project, discuss if the goal of the thesis was reached and if the results are satisfactory.

The first part of this work included the implementation of a spiking neural network, that is comprising a mechanism to keep track of the head direction of a robot. Although this was already done in an earlier project, which served also as a template, the process of implementing it turned out to be the most difficult and time consuming of the whole project. I attribute that to the breakdown of the first ROLLS chip on one hand and to the apparently big device mismatch of the second one on the other. The mismatch, meaning the different behavior of the neuron and synapse circuits caused, that I had to carefully select the neurons in use, adjust the synaptic connections and fine tune the chip biases in order to stabilize the activity in the HD population, but still allow the activity shift trough the whole neuron group. This process, however arduous it may have been, meant that I had to deal intensively with the network structure and the functionality of the ROLLS chip, which helped me with the rest of the project. To that, I implemented additional software to overcome the problems, such as the neuron vector structure, which is hopefully useful for other people working with the same setup in the future.
Because of the little bias-tuning window in which the system worked, there was no space for improving the regularity or consistency of the shifting mechanism. The consequences can be observed in figure 3.4. The activity does not stay for the same time at every neuron and even skips a neuron from time to time, also this behavior can change in every trial. This led to the fact, that keeping track of the head direction is not very precise and can totally fail for single trials. Although the mean of the absolute shift error over all trials is zero, the standard deviation grows with every robot turn and is about 55° after 6 turns, which means that averagely, the head direction of the robot was estimated wrong by 55° in terms of the 360° view.

The other rather big step of the project was to implement the learning behavior between the LED neurons and the IH population. To realize learning, I added a mechanism in the software, which changed the chip biases involved in learning on different keyboard inputs. Biologically, it would make more sense to implement a continuous learning behavior, which potentiates the plastic synapses of the postsynaptic neuron over time if presynaptic spikes are arriving and depressed them again if no presynaptic spikes are incoming. But the missing precision of the shifting mechanisms didn't allow this more realistic learning behavior, because it would require the activity to be exact at the same HD neurons during the same robot orientations for at least two turns. The results show, that adding the learning of the positions from the visual landmarks and correction trough them, improve the precision of the network. The absolute error spread gets noticeably smaller as the standard deviation of the shift error drops each time adding a light about the half and remains almost the same for 3 LED's as figure 3.11 shows.
About a quarter of all trials with LED lights show a distortion of the precision trough either not recognizing all lights in every turn or some wrong learning behavior. A third of these distortions come from not recognized lights. The detection of the blinking frequency is not absolutely reliable, because the LED populations are only stimulated, when enough eDVS events are classified as belonging to a certain blinking frequency. This lead to rare occasions, when not enough eDVS events were classified right and the light is therefore not detected.
The somehow wrong learning occurs at two thirds of all disturbed trials. This includes on one hand that not learning at all took place, but more often, that the LED neurons formed a connection to too much IH neurons and after the light is recognized again, the activity reset is ambiguous.
When evaluating only the trials without disturbing factors, especially the relative error spread gets smaller as figures 3.12, 3.13 show. The absolute spread though converges earlier, but after 6 turns shows only a little improvement for all LED numbers in use. This makes sense for the cases of missing light detections because no reset of the activity in one turn can make a big difference in the shift error for this turn, but eventually the light will be detected again and after all turns the absolute shift error can still be corrected. In the case of ambiguous learning, its not so clear why the absolute shift error is not disturbed more. If there were made multiple connections from a LED group to different IH neurons, the system has probably a tendency to reset the activity still to the correct neuron because this has the strongest connection, as it was the one spiking when the LED neurons

where spiking too.

So the results without the disturbed trials should represent the minimal error spread, the system can achieve. When using two lights as landmarks, the network has an average shift error of about 12°, which seems to be the best achievable result. That this result is not lower has multiple reasons. First, the IH neurons which learned the connection to the LED neurons, spike for a different amount of time, every time the corresponding light is detected, which can result in slightly different shifts of the activity hill relative to the actual orientation of the robot (see figure 3.8). Also can shift errors already accumulate between the time a LED is recognized and the turn is finished, because the underlying shift mechanisms is, as discussed before, not precise. Second, 12° correspond to roughly one neuron, so the average error spread amounts to one neuron. The precision of the system is also limited by the amount of neurons used to model the HD population and therefore the 360° view. For example, if the HD population was modeled with double the amount, 72 neurons, then a shift error of one neuron would correspond only to 5° of the horizontal view and the system would have doubled precision.
As the ROLLS chip comprises just 256 neurons and for every HD neuron there are three additional ones for the integrated IH and the two shift populations, there is no possibility to increase the number of neurons representing the 360° view significantly. This means, on a neuromorphic device with more neurons, the correction mechanisms and the overall precision of the network could be improved by splitting the view up to more neurons. Also if it had less device mismatch, the bias-tune window mentioned before would probably be bigger because the shift would go trough the network easier. This would enable to tune a more stable learning behavior and a slower activity shift, which would also contribute to an increased precision. But with the amount of neurons on the ROLLS chip and the mismatch on this particular chip, the achieved results of the correction trough the visual landmarks are satisfactory and can probably not be improved further.

5. Conclusion

In this work it was shown, that a head direction network to integrate the horizontal orientation of a robot can be implemented on neuromorphic hardware with a neural network using ring attractor dynamics. Additionally, the network is able to learn the positions of landmark and reset its internal representation of the orientation according to the position of these landmarks, which is an important part of orientation in a new environment. A form of path integration and map formation is performed, as the system keeps track of the robots head direction and learns the position of surrounding lights. To further improve the system, next steps would be to increase the numbers of neurons used for the HD, IH and shift populations, which would increase the precision of the system and to implement a more self-reliant learning mechanism, which does not have to be triggered externally. With this, the network would have to make the decision, when it should learn the position of a new landmark and when it should reset its orientation according to this landmark. It should also be able to unlearn the positions of the lights again, if they have not been recognized for a longer period. Maybe, this could be realized, if the connections from the IH to LED population and in the other direction are implemented simultaneously, but with different connection weights. Also, the connection from LED to IH population should gradually get weaker if the LED has not been seen for a longer time. So if the light is detected regularly, the LED $\rightarrow$ IH connection would be stronger and could correct the activity shift, but when the LED spikes were missing for some time or the light is spotted for the first time, the IH $\rightarrow$ LED connection would be stronger and could (re)set the LED $\rightarrow$ IH connection by activate the LED neurons from the current IH neuron, which would again potentiate this new LED $\rightarrow$ IH connection. This would lead to a fast learning of new positions, if the connection LED $\rightarrow$ IH is weak, and would allow to remember the position and reset the activity shift, when the light is seen regularly. Another step to a head direction network suitable for real-world applications would be an implementation of different shifting speeds of the activity hill corresponding to different motor speeds of the robot.

With these possible future improvements, the network could be combined with other neuronally inspired orientation systems, such as a 2D map formation network, to achieve a fully functioning neuromorphic SLAM system.

A. Appendix

A.1 Software code

A.1.1 main.cpp

```
#include <ROLLSListener.h>
#include <ROLLSListener_Log.h>
#include <controller.h>

#include <boost/filesystem.hpp>
#include <boost/date_time.hpp>
#include <boost/program_options.hpp>
#include <TCPConnector.h>
#include <signal.h>  // signals for program abort
#include <iostream>

namespace po = boost::program_options;

// Atomic flag to react to system signals (to abort program
   ↪ )
volatile sig_atomic_t abort_flag = 0;
void aborter(int sig) {  abort_flag = 1; }

int main(int argc, char *argv[]) {
  // Program Options
  // ---------------
  // Variables that will store parsed values.
  std::string robot_address = "192.168.1.42";
  unsigned int robot_port = 56000;

  std::string config = "";

  bool setup_needed = false;
  bool enable_config = false;

  double speed = 1;
  double turn = 1;

  std::string log_dir = "./logs";
  std::string exp_name = "";
  std::string wta_log = "";
  bool no_timestamp = false;
```

```
// Command line args
po::options_description desc("Pushbot Obstacle Avoidance
    ↪ .\n\nOptions");
desc.add_options()
("help", "produce help message")
("robot_address",
 po::value<std::string>(&robot_address)->default_value(
     ↪ robot_address),
 "ip address of robot")
("robot_port",
 po::value<unsigned int>(&robot_port)->default_value(
     ↪ robot_port),
 "port of robot")
("config",
 po::value<std::string>(&config)->default_value(config),
 "set up rolls architecture with given configuration file
     ↪ ")
("speed", po::value<double>(&speed)->default_value(speed)
    ↪ ,
 "factor to multiply speed")
("turn", po::value<double>(&turn)->default_value(turn),
 "factor to multiply turning speed")
("setup", po::bool_switch(&setup_needed),
 "set up rolls architecture")
  ("logdir", po::value<std::string>(&log_dir)->
      ↪ default_value(log_dir),
 "directory to store logs")
("exp", po::value<std::string>(&exp_name)->default_value(
    ↪ exp_name),
 "name of experiment (used for logfiles)")
("no-timestamp", po::bool_switch(&no_timestamp),
 "disable automatic adding of timestamp to experiment
     ↪ name")
;

// Parse input
po::variables_map vm;
po::store(po::parse_command_line(argc, argv, desc), vm);
po::notify(vm);

// Help message
if (vm.count("help")) {std::cout << desc << "\n"; return
    ↪ 1;}

//opens config file if necessary
if (config.empty()) {
  std::cout << "Configuration file name missing!\n";
}else{
  enable_config = true;
}
std::ifstream stream;
if(enable_config){
  stream.open(config);
  if (!stream.good()) {
```

```
      std::cout << "Configuration file not found or not
         ↪ accessible!\n";
      return -1;
    }
  }

    bool enable_logging = ((exp_name.empty() && wta_log.
       ↪ empty()) ? false : true);
    // Prepare Log directory and lof filenames
     if (log_dir.back() !='/') log_dir += '/';
     boost::filesystem::path dir(log_dir);
     if(!(boost::filesystem::exists(dir))){
  std::cout << "Creating directory '" << log_dir << "'..."
     ↪ << std::endl;;
        if (boost::filesystem::create_directory(dir))
      std::cout << "Done." << std::endl;
  };
  if (!no_timestamp) exp_name += "_" + to_iso_string(pt::
     ↪ second_clock::local_time());
    std::string edvs_log = log_dir + exp_name + "_edvs.csv
       ↪ ";
    std::string imu_log = log_dir + exp_name + "_imu.csv";
    std::string rolls_log = log_dir + exp_name + "_rolls.
       ↪ csv";
    std::string robot_log = log_dir + exp_name + "_robot.
       ↪ csv";

  // Initialize Controller, Robot and ROLLS
  // ----------------------------------------

  // Try to connect to robot
  std::cout << "Connecting to robot over TCP on " <<
     ↪ robot_address
            << ":" << robot_port << "...\n";
  int file_descriptor =
    getTCPFileDescriptor(robot_address.c_str(), robot_port)
       ↪ ;

  // Pushbot: Motors have to be enabled, Blink LED to show
     ↪ connection
  PushBot pushbot(file_descriptor);
  pushbot.enableMotors();
  pushbot.switchLED(true);
  usleep(100000);
  pushbot.switchLED(false);
// std::cout << pushbot.twitch() << "\n";
  std::cout << "Robot initialized.\n";

  // eDVS: Set event format when initialized
  EDVSDevice eDVS(file_descriptor);
  if (eDVS.isInit() < 0) {
    std::cout << "eDVS device could not be initialised.\n";
    return -1;
  }
  eDVS.changeEventFormat(EventFormat::EVT_FORMAT_E3);
  std::cout << "eDVS initialized.\n";

  ROLLSDevice* rolls = new ROLLSDevice({0,150,151});
```

```
  std::cout << "ROLLS initialized.\n";

  TerminalInput termInput;
  std::cout << "Terminal device initialized\n";

  //start time for all loggers
  boost::posix_time::ptime start = boost::posix_time::
     ↪ microsec_clock::local_time();

  // Set up logs for rolls
  // Important! Create them in the main scope
  ROLLSListener_Log *rolls_logger;
  //EDVSListener_Log *edvs_logger;
  if (enable_logging)
  {
     rolls_logger = new ROLLSListener_Log(rolls_log, start)
        ↪ ;
     rolls->registerListener(rolls_logger);
     //edvs_logger = new EDVSListener_Log(edvs_log, start);
     //eDVS.registerListener(edvs_logger);
  }
if(setup_needed) rolls->withCodeSetup = true;
 // ROLLS: bad neurons are 0,32, 84 (TODO: Put into config)
  std::cout << "Connecting synapses...\n";
  if(enable_config){
    if (!rolls->loadConfig(stream)) {
      std::cout << "Configuration error.\n";
      return -1;
    }
  }

  Controller controller(rolls, &pushbot);

  //Setting up connection not possible with minijson
 if (setup_needed) {
      std::cout<<"Synapse setup started."<<std::endl;
      controller.apply(*rolls);
      std::cout<<"Synapse setup done."<<std::endl;
   stream.seekg(0);
   rolls->loadConfig(stream);
  }
  std::cout << "Controller initialized." << std::endl;

  // Reset input stream to use again for controller
  stream.seekg(0);

 if(enable_config){
   if (!controller.loadConfig(stream)) {
     std::cout << "Configuration error.\n";
     return -1;
  }
 }

  // register signal handler to be able to stop the program
  signal(SIGINT, aborter);

    // Controller: Register as eDVS- and ROLLS-Listener
```

```
  eDVS.registerListener(&controller);
  rolls->registerListener(&controller);
  termInput.registerListener(&controller);

  // Start listening for events
  eDVS.listen();
  rolls->listen();
  termInput.listen();

  eDVS.startIMU(EDVSIMUEvent::IMUType::IMU_HEADING, IMURate
     ↪ );

  std::cout << "Control loop running..." << std::endl;

  // Keep alive until abort signal is caught
  while (1) {
    usleep(1);
    if (abort_flag) { std::cout << "Stopping program...\n";
       ↪  break; }
  }

  rolls->stopListening();
  eDVS.stopListening();
  termInput.stopListening();
  eDVS.close();
  std::cout << "Connections closed.\n";

  // Wait a little, sometimes the devices need some time to
     ↪  shut down
  usleep(EndDelay);

  std::cout << "Goodbye!" << std::endl;
  return 0;
}
```

A.1.2 controller.h

```
#include <signal.h>  // signals for program abort

#include <boost/filesystem.hpp>
#include <boost/date_time.hpp>
#include <boost/program_options.hpp>
namespace po = boost::program_options;

#include <fstream>
#include <iostream>
#include <string>
#include <array>
#include <vector>
#include <aerctl.h>
#include <unistd.h>

#include <ROLLSArchitecture.h>
#include <PushBot.h>
#include <EDVSDevice.h>
#include <ROLLSDevice.h>
#include <ROLLSListener.h>
#include <ROLLSListener_Log.h>
```

```
#include <RobotListener_Log.h>
#include <EDVSListener_Log.h>
#include <TerminalInputDevice.h>
#include <TerminalInputListener.h>


#include "minijson_reader.hpp"
namespace json = minijson;

struct ROLLSInput { unsigned int neuron; unsigned int
   ↪ synapse; };

struct EventBlinkingLedTracker
    {
        unsigned int x,y,t;
        int p;
    };

const int IMUSpikes = 200;  // Number of spikes for maximum
   ↪  IMU stimulation
const int IMURate = 200;     // IMU sampling rate
const int EndDelay = 200 * 1000;  // Delay for devices to
   ↪ shut down in the end

int LED1_events = 0;
int LED2_events = 0;
int LED3_events = 0;

class Controller : public EDVSListener,
                   public ROLLSListener,
                   public TerminalInputListener,
                   public ROLLSArchitecture{

  public:
   Controller(ROLLSDevice* _rolls, PushBot* _pushbot);

   int speed = 25;

   NeuronVector Drive_left;
   NeuronVector Drive_right;
   NeuronVector Shift_countercwise;
   NeuronVector Shift_cwise;
   NeuronVector Heading;
   NeuronVector IH;
   NeuronVector LED1;
    NeuronVector LED2;
   NeuronVector LED3;
  int testTreshold;

  void receivedNewEDVSEvent(EDVSEvent& e);
  void receivedNewEDVSIMUEvent(EDVSIMUEvent& ev);
  void receivedNewROLLSEvent(unsigned int event);
  void receivedNewTerminalInputEvent( char read_char);
  bool loadConfig(std::ifstream &configstream);

 private:
    // Spike-controlled Pushbot and ROLLS
    ROLLSDevice *rolls;
```

```
  PushBot *pushbot;

  std::vector<ROLLSInput> IMUMapLeft;
 std::vector<ROLLSInput> IMUMapRight;

  void create_architecture();
  void parse_motor_setting(json::istream_context* ctx,
     ↪ json::value const& v);
  void parse_dvs_setting(json::istream_context* ctx,
                         json::value const& v);
  void parse_imu_setting(json::istream_context* ctx,
                         json::value const& v);
  void testNeurons(NeuronGroup testGroup);
  void testNeurons(NeuronVector testVector);
void testNeuron(int neuron);
void testNeuron(NeuronVector group, int neuron);
void testNeuronsKernelOne(NeuronGroup testGroup);
void LED_event(EDVSEvent& e, int LED_numbern, int
   ↪ settings[], std::list<float> &m_xCs, std::list<
   ↪ float> &m_yCs,std::vector<std::list<unsigned int>>
   ↪ &m_t );
void drive_left();
void drive_right();

void learn();
void unlearn();
void stopLearn();

float                          m_xC=64;        // center X
float                          m_yC=64;        // center Y
unsigned int                   m_tC=0;        // center T

unsigned int                   m_minEvtNeigh=1;
unsigned int                   m_minSuppPer=1;
unsigned int                   m_maxEvtDist=8;

unsigned int                   m_mPeriod=100;  // mean of
   ↪ the period
unsigned int                   m_vPeriod=25;  // variance of
   ↪  the period
unsigned int                   m_medPeriods=3;

int LED1_settings[6];
int LED2_settings[6];
int LED3_settings[6];
int fovea_limit1 = 0;
int fovea_limit2 = 0;
int fovea_limit3 = 0;

  // Memory properties
  unsigned int                 m_rows=128;
  unsigned int                 m_cols=128;

  std::vector< std::list<unsigned int> >  m_t1;
  std::vector< std::list<unsigned int> >  m_t2;
  std::vector< std::list<unsigned int> >  m_t3;
```

```
    std::list<float>            m_xCs1;
    std::list<float>            m_yCs1;

    std::list<float>            m_xCs2;
    std::list<float>            m_yCs2;

    std::list<float>            m_xCs3;
    std::list<float>            m_yCs3;

    bool dl = false;
    bool dr = false;
};
```

A.1.3 controller.cpp

```
#include <signal.h>  // signals for program abort

#include <boost/filesystem.hpp>
#include <boost/date_time.hpp>
#include <boost/program_options.hpp>
namespace po = boost::program_options;

#include <fstream>
#include <iostream>
#include <string>
#include <array>
#include <vector>
#include <aerctl.h>
#include <unistd.h>
#include <controller.h>


#include "minijson_reader.hpp"
namespace json = minijson;


// Controller
// ==========

    /*!
     * @brief       Create the Controller
     *
     * @param       _rolls              Rolls
     * @param       _pushbot            PushBot
     */
Controller::Controller(ROLLSDevice* _rolls, PushBot*
  ↪ _pushbot)
    :rolls(_rolls), pushbot(_pushbot) {
      create_architecture();
}
```

```
void Controller::create_architecture(void){
  std::cout << "Parsing Groups" << std::endl;
 if(!rolls->NeuronGroups.empty()){
     for(NeuronVector group : rolls->NeuronGroups){
       if((group.name.compare("Heading"))==0){
         Heading = group;
      }
     else if((group.name.compare("Drive_left"))==0){
         Drive_left = group;
     }
     else if((group.name.compare("Drive_right"))==0){
         Drive_right = group;
     }
     else if((group.name.compare("Shift_cwise"))==0){
      Shift_cwise = group;
     }
     else if((group.name.compare("Shift_countercwise"))==0){
      Shift_countercwise = group;
     }
     else if((group.name.compare("IH"))==0){
      IH = group;
     }
     else if((group.name.compare("LED1")) ==0){
      LED1 = group;
     }
       else if((group.name.compare("LED2")) ==0){
         LED2 = group;
       }
       else if((group.name.compare("LED3")) ==0){
         LED3 = group;
       }
     }
  }

  std::thread a(&Controller::drive_left, this);
  a.detach();
  std::thread b(&Controller::drive_right, this);
  b.detach();
  /*
  connectPlastic(LED1, IH, true);
  connectPlastic(LED2, IH, true);
  connectPlastic(LED3, IH, true);
*/
  connectPlastic(IH, LED1, true);
  connectPlastic(IH, LED2, true);
  connectPlastic(IH, LED3, true);

}

// eDVS events
void Controller::receivedNewEDVSEvent(EDVSEvent& e) {
  //If the event is in the middle of the DVS camera, check
     ↪ frequency (x and y are swapped)
  if (e.y > 60 && e.y<70 && e.pol != 1)
  {
```

```
    LED_event(e, 1, Controller::LED1_settings, m_xCs1,
      ↪ m_yCs1, m_t1);
    LED_event(e, 2, Controller::LED2_settings, m_xCs2,
      ↪ m_yCs2, m_t2);
    LED_event(e, 3, Controller::LED3_settings, m_xCs3,
      ↪ m_yCs3, m_t3);
  }
}

// eDVS IMU
void Controller::receivedNewEDVSIMUEvent(EDVSIMUEvent& ev)
{

}

// ROLLS to Motor
void Controller::receivedNewROLLSEvent(unsigned int event)
{

}

void Controller::receivedNewTerminalInputEvent( char
  ↪ read_char ) {
std::cout << "Start TerminalInputListener Drive" << std::
  ↪ endl;
//std::cout << "Start Keys" << std::endl;
switch(read_char)
      {

        case 'i' : // Go left for
          pushbot->setLeftVelocity(0);
          pushbot->setRightVelocity(speed);
          dl = true;
          break;

        case 'o' : // Go  for
        //Stop all activity
          pushbot->setLeftVelocity(0);
          pushbot->setRightVelocity(0);
          dl = false;
          dr = false;
          break;

        case 'p' :
          pushbot->setRightVelocity(0);
          pushbot->setLeftVelocity(speed);
          dr = true;
          break;

        case 'y' :
          rolls->readPlasticSynapses(IH);
          break;

        case 'n' :
          testNeuron(20);
          break;

        case 'm' :
          testNeuron(25);
          break;
```

```
        case 'w' :
          testNeurons(Heading);
          break;

        case 'l' :
          learn();
          break;

        case 'u' :
          unlearn();
          break;

        case 's' :
          stopLearn();
          break;
      }
}

// Config File Parsing
bool Controller::loadConfig(std::ifstream &configstream) {
  // Try to parse config
  // How minijson works: (10 min read)
  // https://github.com/giacomodrago/minijson_reader
  bool success = true;  // TODO(Alex): Check failures

  json::istream_context ctx(configstream);
  json::parse_object(ctx, [&](const char* key, json::value
     ↪ value) {
        json::dispatch(key)
        <<"pushbot">> [&] {
            json::parse_object(ctx,
                                [&](const char* key, json::
                                   ↪ value value) {
                  json::dispatch(key)
                  <<"motor">> [&] {
                    std::cout << "Parsing Motor Settings...\n
                       ↪ ";
                    parse_array(ctx,
                      [&](json::value v) {
                        Controller::speed = static_cast<
                           ↪ uint32_t>(v.as_long());
                        std::cout << "ROBOT SPEED: " <<
                           ↪ Controller::speed << std::endl;
                       });
                  }
                  <<"dvs">> [&] {
                    std::cout << "Parsing DVS Settings...\n";
                    std::vector<int> settings;
                    parse_array(ctx,
                      [&](json::value value) {
                      settings.push_back(static_cast<uint32_t
                         ↪ >(value.as_long()));
                                  std::cout << value.as_long
                                     ↪ () << std::endl;
```

```
                    });
                    //Parse parameters for LED lights
                    for (int i = 0; i<6; i++){
                      Controller::LED1_settings[i] = settings
                        ↪ .at(i);
                    }
                    for (int i = 0; i<6; i++){
                      Controller::LED2_settings[i] = settings
                        ↪ .at(6+i);
                    }
                    for (int i = 0; i<6; i++){
                      Controller::LED3_settings[i] = settings
                        ↪ .at(12+i);
                    }
                    //Parse after how many events the LED
                      ↪ groups fire
                    Controller::fovea_limit1 = settings.at
                      ↪ (18);
                    Controller::fovea_limit2 = settings.at
                      ↪ (19);
                    Controller::fovea_limit3 = settings.at
                      ↪ (20);
                  }
                  <<"imu">> [&] {
                    std::cout << "Parsing IMU Settings...\n";
                    parse_array(ctx,
                      [&](json::value v) { parse_imu_setting
                        ↪ (&ctx, v); });
                  }
                  <<json::any>> [&]{ json::ignore(ctx); };
              });
          }
          <<json::any>> [&]{json::ignore(ctx); };
  });

  std::cout << "Done!\n";
  return success;
}

// Config Parser Functions
void Controller::parse_motor_setting(json::istream_context*
  ↪ ctx,
                                       json::value const& v) {
    int neuron;
    json::parse_object(*ctx, [&](const char* key, json::
      ↪ value value) {
        json::dispatch(key)
        <<"neuron">> [&]{ neuron = static_cast<int>(value.
          ↪ as_long()); }
        <<"type">> [&]{
            auto type = value.as_string();
            if (strcmp(type, "speed") == 0) {
            } else if (strcmp(type, "left") == 0) {
            } else {
            }
        }
```

```
        <<json::any>> [&]{ json::ignore(*ctx); };
    });
}
void Controller::parse_dvs_setting(json::istream_context*
    ↪ ctx,
                                   json::value const& v) {
    int x, y;
    ROLLSInput input;

    json::parse_object(*ctx, [&](const char* key, json::
        ↪ value value) {
        json::dispatch(key)
        <<"x">> [&]{ x = static_cast<int>(value.as_long());
            ↪ }
        <<"y">> [&]{ y = static_cast<int>(value.as_long());
            ↪ }
        <<"neuron">> [&]{
            input.neuron = static_cast<int>(value.as_long()
                ↪ ); }
        <<"synapse">> [&]{
            input.synapse = static_cast<int>(value.as_long
                ↪ ()); }
        <<json::any>> [&]{ json::ignore(*ctx); };
    });
}
void Controller::parse_imu_setting(json::istream_context*
    ↪ ctx,
                                   json::value const& v) {
    bool left = false;
    ROLLSInput input;

    json::parse_object(*ctx, [&](const char* key, json::
        ↪ value value) {
        json::dispatch(key)
        <<"neuron">> [&]{
            input.neuron = static_cast<int>(value.as_long()
                ↪ ); }
        <<"synapse">> [&]{
            input.synapse = static_cast<int>(value.as_long
                ↪ ()); }
        <<"type">> [&]{ left = (strcmp(value.as_string(), "
            ↪ left") == 0); }
        <<json::any>> [&]{ json::ignore(*ctx); };
    });

    if (left) {
        IMUMapLeft.push_back(input);
    } else {
        IMUMapRight.push_back(input);
    }
}


void Controller::testNeurons(NeuronGroup testGroup){
  for(int n = testGroup.start; n<= testGroup.end; n++)
```

```
  {
     std::cout << "Stimulating Neuron: " << n << std::endl;
    for(int i = 0; i<30; i++)
    {
      for(int j = 0; j<4; j++)
      {
        rolls->stimulate(n, j);
      }
    }
    usleep(500000);
  }
}

void Controller::testNeurons(NeuronVector testVector){
  for(int n : testVector.neurons)
  {
     std::cout << "Stimulating Neuron: " << n << std::endl;
    for(int i = 0; i<30; i++)
    {
      for(int j = 0; j<4; j++)
      {
        rolls->stimulate(n, j);
      }
    }
    usleep(500000);
  }
}

void Controller::testNeuron(int neuron){
  neuron += testTreshold;
  std::cout << "Stimulating Neuron: " << neuron << std::
     ↪ endl;
  for(int i = 0; i<50; i++)
  {
    for(int j = 0; j<4; j++)
    {
      rolls->stimulate(neuron, j);
    }
  }
}

void Controller::testNeuron(NeuronVector group, int neuron)
   ↪ {
  neuron = group.neurons.at(neuron+testTreshold);
 std::cout << "Stimulating Neuron: " << neuron << std::endl
    ↪ ;
 for(int i = 0; i<50; i++)
 {
    for(int j = 0; j<4; j++)
    {
       rolls->stimulate(neuron, j);
    }
 }
}

void Controller::testNeuronsKernelOne(NeuronGroup testGroup
   ↪ ){
```

```
  for(int neuron = testGroup.start; neuron <= testGroup.end
     ↪ ; neuron++)
  {
    std::cout << "Stimulating Neurons: " << neuron-1 << " "
       ↪  << neuron << " " << neuron+1 << std::endl;
    for(int i = 0; i<10; i++)
    {
      for (int j = 0; j<4; j++)
      {
        rolls->stimulate(neuron-1, j);
        rolls->stimulate(neuron, j);
        rolls->stimulate(neuron+1, j);
      }
    }
    usleep(1000000);
  }
}
void Controller::LED_event(EDVSEvent& e, int LED_number,
   ↪ int settings[], std::list<float> &m_xCs, std::list<
   ↪ float> &m_yCs, std::vector<std::list<unsigned int>> &
   ↪ m_t){
  m_minEvtNeigh=settings[3];
  m_minSuppPer=settings[4];
  m_maxEvtDist=settings[5];

  m_mPeriod=settings[0];  // mean of the period
  m_vPeriod=settings[1];  // variance of the period
  m_medPeriods=settings[2];
  {
    EventBlinkingLedTracker evt;
    evt.x = e.y;
    evt.y = e.x; // mounted the wrong way around
    evt.t = e.timestamp;
    evt.p = e.pol;
    //if polarity is on-event ignore
    const int r = 5;

    //bottom = maximum between 0 and  (event -5 ) so if
       ↪ event is between 0-5 it will return 0
    //same thing for the top except

    //I'm assuming that m-row and m_cols are the max number
       ↪  of columns and rows of the DVS
    //if event number x = 50 y = 46 comes in --> bRow = 45
       ↪ tRow= 55    bCol = 41 tCol = 51

    int bRow = std::max(0,(int)evt.x-r);
    int tRow = std::min((int)m_rows,(int)evt.x+r);

    int bCol = std::max(0,(int)evt.y-r);
    int tCol = std::min((int)m_cols,(int)evt.y+r);
    //row is x
    //col is y

    //define a bunch of stuff
```

```
float meanX = 0, meanY = 0, meanT = 0;
m_t.resize(m_rows*m_cols);  // timestamps

unsigned int nEvts = 0, nEvtsPrev = 0, nEvtsDist = 0;

std::list<unsigned int>::iterator oldEvtsBeg;
unsigned int nEvtPer2 = 0, nEvtsDist2 = 0;
//unsigned int nEvtPer3 = 0, nEvtsDist3 = 0;

//for an event in the cycle through the 10 event window
  ↪ in rows/column
for(int i=bRow; i<tRow; i++) //10
{
  for(int j=bCol; j<tCol; j++)//10
  {
    std::list<unsigned int>* ts = &(m_t[i*m_cols+j]);
    oldEvtsBeg = ts->end();
    for(std::list<unsigned int>::iterator itTs = ts->
      ↪ begin(); itTs != ts->end(); itTs++)
    {
      unsigned int dt = evt.t - *itTs;
      if( dt <= m_vPeriod )
      {
          meanX += i;
          meanY += j;
          meanT += *itTs;
          nEvts++;
      }
      //between 1 vPeriod
      else if(    (dt >= m_mPeriod - m_vPeriod)
                && (dt <= m_mPeriod + m_vPeriod)
              )
      {
          nEvtsPrev++;
      }
      else if(    (dt <= m_mPeriod - 1.1*m_vPeriod)
                && (dt >= 2*m_vPeriod)
              )
      {
          nEvtsDist++;
      }

      else if(    (dt >= 2*m_mPeriod - m_vPeriod)
                && (dt <= 2*m_mPeriod + m_vPeriod)
              )
      {
          nEvtPer2++;
      }
      else if(    (dt >= m_mPeriod + 1.1*m_vPeriod)
                && (dt <= 2*m_mPeriod - 1.1*m_vPeriod)
              )
      {
          nEvtsDist2++;
      }

      else if ( dt > 3*(m_mPeriod + m_vPeriod) )
```

```
            {
                oldEvtsBeg = itTs;
                break;
            }
        }
        ts->erase(oldEvtsBeg,ts->end());
    }
}
// Add the event to the list
m_t[evt.x*m_cols+evt.y].push_front(evt.t);
// if I am supported by the previous period
// that there are enough events in my neighbourhood
// and not too many distractors
if(     (nEvtsPrev      >    m_minSuppPer)
  &&   (nEvts          >    m_minEvtNeigh)
  &&   (nEvtsDist      <    m_maxEvtDist)
  &&   (nEvtPer2       >    m_minSuppPer)
  &&   (nEvtsDist2     <    m_maxEvtDist)
  )
{
    meanX /= nEvts;
    meanY /= nEvts;
    meanT /= nEvts;

    m_xC = meanX;
    m_yC = meanY;
    m_tC = meanT;

    m_xCs.push_front(m_xC);
    if(m_xCs.size() > m_medPeriods)
        m_xCs.pop_back();

    std::vector<float> xCs(m_xCs.begin(),m_xCs.end());
    int nX = xCs.size()/5; // 4? 20/5?
    std::nth_element(xCs.begin(),xCs.begin()+nX,xCs.end
      ↪ ());

    m_xC = xCs[nX];// take the fitth element?

    m_yCs.push_front(m_yC);
    if(m_yCs.size() > m_medPeriods)
        m_yCs.pop_back();

    std::vector<float> yCs(m_yCs.begin(),m_yCs.end());
    int nY = yCs.size()/5;
    std::nth_element(yCs.begin(),yCs.begin()+nY,yCs.end
      ↪ ());

    m_yC = yCs[nY];

   switch (LED_number){
     case 1: LED1_events++;
       if (LED1_events == 5){
         std::cout << LED_number << std::endl;
         for (int ll = 0; ll < 4; ll++)    {
           for (int j = 1; j < 4; j++) {
             rolls->stimulate(LED1, j);
           }
```

```
            }
            LED1_events =-fovea_limit1;
            LED2_events =0;
            LED3_events =0;
          }
        break;
        case 2: LED2_events++;
          if (LED2_events == 5){
          std::cout << LED_number << std::endl;
            for (int ll = 0; ll < 4; ll++)    {
              for (int j = 1; j < 4; j++) {
                rolls->stimulate(LED2, j);
              }
            }
            LED1_events =0;
            LED2_events =-fovea_limit2;
            LED3_events =0;
          }
        break;
        case 3: LED3_events++;
          if (LED3_events == 5){
          std::cout << LED_number << std::endl;
            for (int ll = 0; ll < 4; ll++)    {
              for (int j = 1; j < 4; j++) {
                rolls->stimulate(LED3, j);
              }
            }
            LED1_events =0;
            LED2_events =0;
            LED3_events =-fovea_limit3;
          }
        break;
      }
    }
  }
}

void Controller::drive_left(){
  while(true)
  {
    while(dl)
    {
      for(int i = 0; i<10; i++)
      {
        for(int j = 0; j<4; j++)
        {
          rolls->stimulate(Drive_left,j);
        }
      }
      usleep(100000);
    }
    usleep(200000);
  }
}
void Controller::drive_right(){
  while(true)
```

```
  {
    while(dr)
    {
      for(int i = 0; i<10; i++)
      {
        for(int j = 0; j<4; j++)
        {
          rolls->stimulate(Drive_right,j);
        }
      }
      usleep(50000);
    }
    usleep(200000);
  }
}

//setting the ROLLS biases for learning
void Controller::learn(){
  Bias SL_THDN = {.id = 13, .coarse = 7, .fine = 100, .high
    ↪  = 1, .type = 0, .cascode = 0, .enable = 1};
  Bias SL_MEMTRH = {.id = 14, .coarse = 6, .fine = 20, .
    ↪ high = 1, .type = 1, .cascode = 0, .enable = 1};
  Bias PA_WDRIFTDN = {.id = 50, .coarse = 6, .fine = 140, .
    ↪ high = 1, .type = 1, .cascode = 0, .enable = 1};
  Bias PA_DELTADN = {.id = 53, .coarse = 5, .fine = 113, .
    ↪ high = 1, .type = 1, .cascode = 0, .enable = 1};
  Bias SL_THMIN = {.id = 16, .coarse = 7, .fine = 200, .
    ↪ high = 1, .type = 1, .cascode = 0, .enable = 1};
  aerSetBias(&SL_THDN);
  aerSetBias(&SL_MEMTRH);
  aerSetBias(&PA_WDRIFTDN);
  aerSetBias(&PA_DELTADN);
  aerSetBias(&SL_THMIN);
}

//biases for unlearning
void Controller::unlearn(){
  Bias SL_THDN = {.id = 13, .coarse = 0, .fine = 100, .high
    ↪  = 1, .type = 0, .cascode = 0, .enable = 1};
  Bias SL_MEMTRH = {.id = 14, .coarse = 0, .fine = 238, .
    ↪ high = 1, .type = 1, .cascode = 0, .enable = 1};
  Bias PA_WDRIFTDN = {.id = 50, .coarse = 0, .fine = 140, .
    ↪ high = 1, .type = 1, .cascode = 0, .enable = 1};
  Bias PA_DELTADN = {.id = 53, .coarse = 0, .fine = 113, .
    ↪ high = 1, .type = 1, .cascode = 0, .enable = 1};
  Bias SL_THMIN = {.id = 16, .coarse = 7, .fine = 200, .
    ↪ high = 1, .type = 1, .cascode = 0, .enable = 1};
  aerSetBias(&SL_THDN);
  aerSetBias(&SL_MEMTRH);
  aerSetBias(&PA_WDRIFTDN);
  aerSetBias(&PA_DELTADN);
  aerSetBias(&SL_THMIN);
}

void Controller::stopLearn(){
```

```
  Bias SL_THMIN = {.id = 16, .coarse = 0, .fine = 200, .
     ↪ high = 1, .type = 1, .cascode = 0, .enable = 1};
  aerSetBias(&SL_THMIN);
}
```

A.1.4 NeuronVector

```
struct NeuronVector{
public:
 NeuronVector(){};
 NeuronVector(std::string name, std::vector<int> neurons){
  this->neurons = neurons;
  start = neurons.front();
  end = neurons.back();
  this->name = name;
 }
 std::vector<int> neurons;
 int start;
 int end;
 std::string name;

 int operator[] (int i) {
  return neurons[i];
 }

 std::vector<int>::iterator getIt(){
  std::vector<int>::iterator it = neurons.begin();
  return it;
 }

 int size(){
  return neurons.size();
 }

 void pushback(int i){
  neurons.push_back(i);
 }
};
```

A.2 User Guide

Quick start user guide for working with the ROLLS processor and Pushbot

David Niederberger

June 25, 2018

Contents

1 Introduction

This user guide contains a step-by-step beginners tutorial on how to work with the ROLLS, Parallella board and Pushbot. Further it includes some code documentations and explanations of the software architecture and helper classes which I see as important to get started with a project. If you want to learn more about the hardware properties of the ROLLS chip or Parallella board, I recommend you the ROLLS and Parallella User Guide written by Dora Sumislawska.

2 Step-by-step guide to set up a new project

2.1 Installation of different libraries

In order to compile the c++ code in your project, some additional libraries are needed. On the parallella board, they are already installed and it would be possible to compile the code only on the board, however I strongly recommend compile your code first on your computer as it takes the Parallella much longer to compile than the average PC/laptop. These are the libraries you have to install on your computer:

2.1.1 Boost

To install Boost on Ubuntu, just type sudo apt-get install libboost-dev into the command line. If you work with another Linux Distribution, google the installation process.

2.1.2 OpenCV

This is a rather complicated installation process and depends on your distribution. If you work on Ubuntu, I can recommend this installation guide: https://www.pyimagesearch.com/2016/10/24/ubuntu-16-04-how-to-install-opencv/

2.1.3 CMake

Again if you work on Ubuntu, type sudo apt-get install cmake into the command line, else google the installation process.

2.2 Download the project structure from GitLab

You can either download the repository or "clone" it. The advantage of cloning the repository to your computer is, that you can upload your new or changed code to GitLab again. There are many reasons why you should work with GitLab, especially if other people are working with the same code simultaneously. If you are new to GitLab, its worth to read up on it on this website: https://docs.gitlab.com/ee/gitlab-basics/
To obtain the project structure, open this GitLab repository:
https://code.ini.uzh.ch/davidn/NCR_lib_new and either download it or clone it.

2.2.1 Clone a repository

- Open a new terminal and navigate to the directory you want the git repository in.
- Enter `git clone` *`URL`* where URL stands for the URL you find in the HTTPS box on the repository website. If the repository has not changed since this tutorial was written, it is
  https://code.ini.uzh.ch/davidn/NCR_lib_new.git.
- Follow the instructions from the command line

2.3 The project structure

Now that you have obtained everything from GitLab, lets see for what the different folder are.

2.3.1 NCSRobotLib

This folder contains every helper class which is needed to program the ROLLS or Pushbot. If you have to add some functionality to these helper classes, make sure that it works 100% before uploading it to Gitlab again, because the NCSRobotLib is a shared library.

2.3.2 paex-dist-cc

Here you will find all files related to communication with the ROLLS chip. Usually you don't have to modify anything in it.

2.3.3 sample experiment

In this folder, there are all files needed for an experiment with the ROLLS and Pushbot.

include & src At the moment, these folders contain the controller.h and controller.cpp files. This is the corepiece of the software architecture where you can specify how events from the ROLLS, eDVS and keyboard should be handled and configure the behavior of the robot. It also includes various test functions for the ROLLS and some parsing functionality for configuring the ROLLS chip over a JSON file. In the sample experiment folder, these files offer a code skeleton with every method already defined, but empty.

main.cpp In the main class, all important experiment agents are initialized. A connection to the robot is made and listeners for ROLLS, eDVS and keyboard events are instantiated and registered.

build This folder contains the CMakeLists-file, which is used to compile your code.

create_config.py This file creates a JSON configuration file, which can be parsed and used to set up a neural network architecture.

2.3.4 Misc experiments

All other folders contain experiments from other users.

2.4 Create a neural network architecture on the ROLLS

To create and load a neural network on the ROLLS, there are two possibilities. The first one is, to use the create_config.py routine to create a new JSON file, which is parsed to a new architecture. The parsed neuron groups

are stored in the neurongroup-list of the controller class. If you want to access the neuron groups individually, which is very likely, then you have to declare and instantiate your neuron groups in the create_architecture method of the controller class. To see how to do that, look at the example in the controller.cpp file. The create_config.py file is not yet very well-engineered and somewhat messy, so theres room for improvement. In section 3.1.1, there's an explanation of the different routines within it. To see, how the structure of the JSON file is, look at the my_config.json file in the sample_experiment folder.
The second method to create a network is directly in the `create_architecture()` method of the controller.cpp file. Here you can instantiate new neuron groups and connect them with the connection methods from the ROLLSArchitecture class described in section 3.2.2. At first this seems easier than the parsing method, but the drawback of this method is, that you always have to recompile your controller.cpp file. As compiling on the Parallella takes very long compared to a normal laptop/PC, this can get very time consuming. Compared to this, you can change the network structure and connections in the create_config.py file very fast, upload it to the Parallella and just run the experiment without recompiling it. So I recommend using the first way to create the network. The only drawback of this method is, that there is not yet a mechanism to parse plastic synapse connection. But you can use both methods together, so you can declare your whole network structure in the JSON file and only the plastic synapse connections in the controller class.
Important: There are two flags for specifying which creation-methods should be applied. The –config flag sets the parsing method and the –setup flag sets the in-code-method. You can also set both flags. After the –config flag you have to write the name of the JSON configuration file you want to use.
When setting new nonplastic synapse connections, you can choose between four excitatory (1,2,3,4) and four inhibitory (-1,-2,-3,-4) weights. The different weight can be set in the ROLLS web interface. To learn more about this interface, I refer to the tutorial written by Dora Sumislawska.

2.5 Handling events from ROLLS, eDVS and keyboard inputs

In the controller class, you'll find different methods to specify, what should happen when certain events are received. The `receivedNewEDVSEvent(EDV-`

`SEvent& e)` takes an EDVSEvent as an argument which has the fields x, y, pol, and timestamp. These are the coordinates and the polarity of a pixel and a timestamp when the event happened.
`receivedNewEDVSIMUEvent(EDVSIMUEvent& ev)` takes a similar argument, but this event contains the field x, y, z, timestamp and type. The first three are coordinates, again a timestamp and the type specifies which IMU type the event has. These types are HEADING, GYRO, COMPASS and ACCELEROMETER.
The `receivedNewROLLSEvent(int events)` just takes the number of the neuron (0-255) which caused the event, meaning spiked.
In the `receivedNewTerminalInput(char read_char)` you can specify what should happen with keyboard inputs. This is very useful for interacting with the robot and ROLLS.

2.6 Sending spikes to the ROLLS and control the robot

To stimulate the ROLLS from outside, just call `rolls→stimulate(i,j)` from within the controller.cpp file. i specifies, which neuron should be stimulated (0-255) and j specifies over which virtual synapse. j can have four different values (0,1,2,3) but often you should stimulate all of them at least once to stimulate the ROLLS enough for some impact.
To control the Pushbot, you can activate his motors by calling `pushbot→setLeftVelocity(i)` and `pushbot→setRightVelocity(i)` again within the controller.cpp file. Here, i is an integer between 0 and 100. These mechanisms to stimulate the ROLLS and steer the robot can be called as reaction to incoming events, for example can you specify different key inputs to activate and deactivate the Pushbot motors and build a little robot steering system.

2.7 Set up and run the experiment

Now you know how to implement a neural network architecture and set ROLLS and robot behavior. Before you can run your experiment, you have to adapt some code fragments to you specific project. First, in the main class, change, if necessary the robot's IP adress in the string `robot_address`. This depends on the router you are using. To obtain the IP adress, connect to the dedicated network and visit the admin page of the router.
Next, open the CMakeLists.txt file in the build folder and change the line `project (sample_project)` to `project(yourprojectname)` and change ev-

ery occurrence of `sample_exp` to your experiments name. This name will determine the name of the executable file of your experiment.
Now you can compile your experiment from the terminal by navigating into the build folder and first enter `cmake .` and then `make`. If you don't get any compiling errors, its time to load your code to the Parallella board. For this, navigate back to the NCR_lib_new folder and enter the following command: `scp -r yourProjectFolder root@parallella_IP:/home/parallella/code`. The IP address of the Parallella can also be obtained from the router's admin page. With the command `scp` you can also load single files to the Parallella, which you most certainly have to do later in your project. After loading the code to Parallella, you have to compile it again. As discussed before, I recommend you to first compile your code on your machine to check if there are any errors, because compiling on the Parallella board is very time consuming.
Now after compiling your code again on Parallella you can run it by entering `./yourExperimentName' --someOptionalFlag`. Don't forget to set the flags for creating the neural network architecture.

3 Important files and methods

3.1 Parsing

3.1.1 create_config.py

This python routine creates the JSON configuration file for a neural network architecture. You can create neuron groups through lists, how to do this exactly shows an example in the code. Then there are many routines to specify connections. Usually they take as arguments two neuron lists and some other parameters which specify the connection. To find out which connection type a routine implements, read the comment in the code. Once you understand how the connection routines work, you can easily add new ones if needed.
In this file you can also specify variable values for pushbot settings, like motor or eDVS values. You can add any value you want, you just have to add the corresponding parsing function in the loadConfig() method of the controller class. Again, the advantage of parsing variable values rather than specifying them in the code is, that you don't have to recompile the code every time you change the value.

3.1.2 load_config()

This method in the controller class parses the JSON configuration file. You can extend it to parse any program variables you like. The JSON file is parsed through a minijson-reader, how that works is explained on this site: https://github.com/giacomodrago/minijson_reader.

3.2 Creating a neural network within the code

3.2.1 ROLLSDevice

This class contains all methods related to the ROLLS chip, such as stimulate neurons and register a ROLLS listener. Also, here are the structs `NeuronGroup` and `NeuronVector` defined.The difference is, that a neuron group is a continuous list of neurons and for the instantiating a group, you have to specify only the beginning and end neuron, while the neuron vector can contain any neurons but for instantiating it you have to give a list with all contained neurons. I recommend to use the neuron vector, because you are much more flexible with the choice of neurons you want to use. When using the parsing method to create your network, the neuron populations are automatically stored as vectors.

3.2.2 ROLLSArchitecture.cpp

This class contains a lot of methods to connect different neuron populations. Every method takes either `NeuronGroup`'s or `NeuronVector`'s as arguments. The names of the methods are mostly self-explanatory and if you want, you can add new connections methods. The mechanisms to set new connections is fairly simple, there is a 256 x 256 array for nonplastic synapses and one for plastic synapses. To set a new connection, between neuron i and j, you have to set the corresponding array entry to 'true' for plastic synapses and to a weight between -4 and 4 for nonplastic synapses.

3.3 Logging

There is software, which allows you to log any events, which may be received during an experiment. The logging objects are instantiated and registered in the main file. To enable the different loggers, you have to uncomment the lines of code in main.cpp, which are responsible for instantiating and

registering the loggers. If you do that, the log files can be found in the log-folder which is located in the build-folder.

Bibliography

Brader, Joseph M. et al. (2007). "Learning real-world stimuli in a neural network with spike-driven synaptic dynamics". In: *Neural Comput.* 19.11, pp. 2881–2912. ISSN: 0899-7667. DOI: 10.1162/neco.2007.19.11.2881.

Cadena, Cesar et al. (2016). "Past, Present, and Future of Simultaneous Localization and Mapping: Toward the Robust-Perception Age". In: *IEEE Trans. Rob.* 32.6, pp. 1309–1332. ISSN: 1552-3098. DOI: 10.1109/TRO.2016.2624754.

DVS (2015). [Online; accessed 15. Jun. 2018]. URL: http://siliconretina.ini.uzh.ch/wiki/index.php.

Indiveri, Giacomo et al. (2011). "Neuromorphic Silicon Neuron Circuits". In: *Front. Neurosci.* 5. ISSN: 1662-453X. DOI: 10.3389/fnins.2011.00073.

Kreiser, Raphaela et al. (2018). "A Neuromorphic Approach to Path Integration: A Head-Direction Spiking Neural Network with Vision-driven Reset". In: *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*, pp. 1–5. ISSN: 2379-447X. DOI: 10.1109/ISCAS.2018.8351509.

Lichtsteiner, P. et al. (2006). "A 128 X 128 120db 30mw asynchronous vision sensor that responds to relative intensity change". In: *2006 IEEE International Solid State Circuits Conference - Digest of Technical Papers*, pp. 2060–2069. ISSN: 0193-6530. DOI: 10.1109/ISSCC.2006.1696265.

Liu, Shih-Chii and Tobi Delbruck (2010). "Neuromorphic sensory systems". In: *Curr. Opin. Neurobiol.* 20.3, pp. 288–295. ISSN: 0959-4388. DOI: 10.1016/j.conb.2010.03.007.

Markram, Henry et al. (2012). "Spike-Timing-Dependent Plasticity: A Comprehensive Overview". In: *Front. Synaptic Neurosci.* 4. ISSN: 1663-3563. DOI: 10.3389/fnsyn.2012.00002.

Mead, Carver (1989). *Analog VLSI and neural systems*. Addison-Wesley Longman Publishing Co., Inc. ISBN: 978-0-201-05992-2. URL: http://dl.acm.org/citation.cfm?id=64998.

Milde, Moritz B. et al. (2017). "Obstacle Avoidance and Target Acquisition for Robot Navigation Using a Mixed Signal Analog/Digital Neuromorphic Processing System". In: *Front. Neurorob.* 11. ISSN: 1662-5218. DOI: 10.3389/fnbot.2017.00028.

Milford, M. J. et al. (2004). "RatSLAM: a hippocampal model for simultaneous localization and mapping". In: *Robotics and Automation, 2004. Proceedings. ICRA '04. 2004 IEEE International Conference on* 1, 403–408Vol.1. ISSN: 1050-4729. DOI: 10.1109/ROBOT.2004.1307183.

Milford, Michael John (2008). *Robot Navigation from Nature*. Springer, Berlin, Heidelberg. ISBN: 978-3-540-77519-5. DOI: 10.1007/978-3-540-77520-1.

O'Keefe, J. and D. H. Conway (1978). "Hippocampal place units in the freely moving rat: why they fire where they fire". In: *Exp. Brain Res.* 31.4, pp. 573–590. ISSN: 0014-4819. URL: https://www.ncbi.nlm.nih.gov/pubmed/658182.

O'Keefe, J. and J. Dostrovsky (1971). "The hippocampus as a spatial map. Preliminary evidence from unit activity in the freely-moving rat". In: *Brain Res.* 34.1, pp. 171–175. ISSN: 0006-8993. URL: https://www.ncbi.nlm.nih.gov/pubmed/5124915.

Parallella (2018). [Online; accessed 2. Jun. 2018]. URL: https://www.parallella.org/board.

Poon, Chi-Sang and Kuan Zhou (2011). "Neuromorphic Silicon Neurons and Large-Scale Neural Networks: Challenges and Opportunities". In: *Front. Neurosci.* 5. ISSN: 1662-453X. DOI: 10.3389/fnins.2011.00108.

Pushbot (2015). [Online; accessed 2. Jun. 2018]. URL: https://inilabs.com/products/pushbot.

Qiao, Ning et al. (2015). "A reconfigurable on-line learning spiking neuromorphic processor comprising 256 neurons and 128K synapses". In: *Front. Neurosci.* 9. ISSN: 1662-453X. DOI: 10.3389/fnins.2015.00141.

Redish, A. David et al. (1996). "A coupled attractor model of the rodent head direction system". In: *Network: Computation in Neural Systems* 7.4, pp. 671–685. ISSN: 0954-898X. DOI: 10.1088/0954-898X_7_4_004.

Seelig, Johannes D. and Vivek Jayaraman (2015). "Neural dynamics for landmark orientation and angular path integration". In: *Nature* 521.7551, p. 186. ISSN: 1476-4687. DOI: 10.1038/nature14446.

Siciliano, Bruno (2007). *Springer Handbook of Robotics*. Springer-Verlag. ISBN: 978-35402395-NaN. URL: http://dl.acm.org/citation.cfm?id=1209344.

Song, Pengcheng and Xiao-Jing Wang (2005). "Angular path integration by moving "hill of activity": a spiking neuron model without recurrent excitation of the head-direction system". In: *J. Neurosci.* 25.4, pp. 1002–1014. ISSN: 1529-2401. DOI: 10.1523/JNEUROSCI.4172-04.2005.

Taube, J. S. et al. (1990). "Head-direction cells recorded from the postsubiculum in freely moving rats. I. Description and quantitative analysis". In: *J. Neurosci.* 10.2, pp. 420–435. ISSN: 0270-6474. URL: https://www.ncbi.nlm.nih.gov/pubmed/2303851.

Zhang, K. (1996). "Representation of spatial orientation by the intrinsic dynamics of the head-direction cell ensemble: a theory". In: *J. Neurosci.* 16.6, pp. 2112–2126. ISSN: 0270-6474. URL: https://www.ncbi.nlm.nih.gov/pubmed/8604055.

ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Neuronally-inspired path integration and map formation with visual correction realized in mixed-signal neuromorphic devices.

Authored by (in block letters):
For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
Niederberger	David

With my signature I confirm that

- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**
Zürich, 25.6.2018	D. Niederberger

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.